НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

«До захисту допущено»
Завідувач кафедри
_____ О.В. Коваль
(підпис) (ініціали, прізвище)
“___”_____2019р.

Магістерська дисертація

зі спеціальності 121 Інженерія програмного забезпечення
за спеціалізацією Інженерія програмного забезпечення розподілених систем
на тему: Розподілена архітектура серверу на основі serverless та мови запитів GraphQL

Виконав: студент 6 курсу, групи ТВ-з81мп
_____ _____
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник _____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ - 2019

Національний технічний університет України
“ Київський політехнічний інститут ім. Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти другий, магістерський

Спеціальність 121 “ Інженерія програмного забезпечення ”

Спеціалізація Інженерія програмного забезпечення розподілених систем

ЗАТВЕРДЖУЮ

_____ _____
(прізвище, ініціали) (підпис)

«____» _____ 201 р.

ЗАТВЕРДЖУЮ
Завідувач кафедри
Коваль О.В. _____
(прізвище, ініціали) (підпис)

«____» _____ 2017р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ ДИСЕРТАЦІЮ СТУДЕНТУ

Брунько Павло Володимирович
(прізвище, ім’я, по батькові)

1. Тема дисертації Розподілена архітектура серверу на основі serverless та мови запитів GraphQL
Науковий керівник Шаповалова Світлана Ігорівна, кандидат технічних наук, доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від “1” листопада 2019 року № 3812-с
2. Строк подання студентом дисертації “9” грудня 2019 року
3. Об’єкт дослідження – програмне забезпечення розподілених систем
4. Предмет дослідження – архітектура веб-серверу на основі технології Serverless
5. Перелік питань, які потрібно розробити:
- дослідити поняття архітектури веб-серверу та еволюцію підходів побудови середовищ розгортання веб-серверу;
- провести аналіз випадків можливості та доцільності застосування Serverless (безсерверних обчислень) для побудови веб-серверу;
- провести порівняльний аналіз традиційної та Serverless архітектури веб-серверу, виявити переваги та недоліки застосування безсерверних обчислень у веб-середовищі;
- розробити гібридну архітектуру веб-серверу для використання переваг Serverless та традиційної архітектур в одному застосунку;

- реалізувати API веб-серверу за допомогою GraphQL;
- провести обчислювальні експерименти з визначення продуктивності та ефективності застосування трьох підходів до розробки веб-серверу – традиційного, Serverless та гібридної архітектури;
- розробити рекомендації щодо застосування Serverless для побудови веб-серверу.

---

6. Перелік ілюстративного матеріалу 3 додатки, 37 рисунків, 18 таблиць
7. Перелік публікацій
XVII міжнародна науково-практична конференція молодих вчених та студентів «Сучасні проблеми наукового забезпечення енергетики» (Київ, 23-26 квітня 2019 р.)

8. Дата видачі завдання «12» вересня 2019 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Постановка цілей та завдань магістерської дисертації	12 вересня 2018 р. – 10 жовтня 2018 р.	
2	Аналітичний огляд літератури, бази джерел за темою дисертації	10 жовтня 2018 р. – 14 січня 2019 р.	
3	Вивчення питань з методології та методики написання магістерської дисертації	14 січня 2019 р. – 25 січня 2019 р.	
4	Підготовка концепції дисертації	25 січня 2019 р. – 20 лютого 2019 р.	
5	Написання основних розділів автореферату	20 лютого 2019 р.. – 1 березня 2019 р.	
6	Підготовка тезисів доповіді	01-10 березня 2019 р.	
7	Участь у науковій конференції	23-26 квітня 2019 р.	
8	Підготовка основних розділів дисертації	26 квітня 2019 р. – 10 серпня 2019 р.	
9	Підготовка стартап-проекту	10 серпня 2019 р. – 2 вересня 2019 р.	
10	Переддипломна практика	2 вересня 2019 р. – 25 жовтня 2019 р.	
11	Захист програмного продукту	25 жовтня 2019 р.	
12	Попередній захист роботи	22 листопада 2019 р.	
13	Оформлення диплому	22 листопада 2019 р. – 9 грудня 2019 р.	
14	Здача всіх матеріалів на підпис зав кафедрою	9 грудня 2019 р.	
15	Захист магістерської дисертації	Грудень 2019 р.	

Студент _____ _____
(підпис) (прізвище та ініціали)

Науковий керівник _____ _____
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Магістерська дисертація складається зі вступу, чотирьох розділів, висновку, переліку посилань з 34 найменувань, 3 додатки, і містить 37 рисунків, 18 таблиць. Повний обсяг магістерської дисертації складає 97 сторінок, з яких перелік посилань займає 5 сторінок, додатки – 4 сторінки.

Актуальність теми. Розробка архітектури веб-серверу потребує значних зусиль, спрямованих на забезпечення гнучкої інфраструктури середовища розгортання серверу. Крім цього, потрібно врахувати, що значна частина виділених та придбаних обчислювальних ресурсів не використовується протягом більшої частини часу існування серверу. Зазначені причини зумовлюють необхідність дослідження можливості застосування безсерверних обчислень (Serverless) для побудови веб-серверу.

Мета дослідження полягає в розробці архітектурного рішення для побудови веб-серверу на основі технології Serverless.

Для досягнення поставленої мети були сформульовані наступні **завдання**, що визначили логіку дослідження та його структуру:

- дослідити поняття архітектури веб-серверу та еволюцію підходів побудови середовищ розгортання веб-серверу;
- провести аналіз випадків можливості та доцільності застосування Serverless (безсерверних обчислень) для побудови веб-серверу;
- провести порівняльний аналіз традиційної та Serverless архітектури веб-серверу, виявити переваги та недоліки застосування безсерверних обчислень у веб-середовищі;
- розробити гібридну архітектуру веб-серверу для використання переваг Serverless та традиційної архітектур в одному застосунку;
- реалізувати API веб-серверу за допомогою GraphQL;

- провести обчислювальні експерименти з визначення продуктивності та ефективності застосування трьох підходів до розробки веб-серверу – традиційного, Serverless та гібридної архітектури;
- розробити рекомендації щодо застосування Serverless для побудови веб-серверу.

Об’єктом дослідження є програмне забезпечення розподілених систем.

Предметом дослідження є архітектура веб-серверу на основі технології Serverless.

Методи дослідження. Розв’язання поставлених задач виконувались з використанням наступних засобів:

- програмного засобу для створення та редагування діаграм MS Visio для візуалізації запропонованої архітектури;
- сервісів здійснення експериментів з продуктивності веб-додатку autocannon, blazemeter.com, jmeter для оцінки характеристик запропонованої архітектури;
- мови запитів GraphQL для побудови API веб-серверу.

Наукова новизна одержаних результатів полягає в тому, що набуло подальшого розвитку структурне рішення для розробників веб-серверів, яке дозволяє спростити розробку та зменшити витрати на обслуговування додатку шляхом використання технології Serverless.

Практичне значення одержаних результатів роботи полягає в розробці гібридної архітектури веб-серверу з використанням Serverless та GraphQL, а також виявлення переваг та недоліків застосування гібридної та Serverless архітектури.

Апробація результатів дисертації. Основні положення роботи доповідались і обговорювались на XVII міжнародній науково-практичній конференції молодих вчених та студентів «Сучасні проблеми наукового забезпечення енергетики» (Київ, 23-26 квітня 2019 р.):

Публікації. Наукові положення дипломної роботи опубліковані в 1 роботі.

Ключові слова. *ВЕБ-СЕРВЕР, АРХІТЕКТУРА, WEB, SERVERLESS, ПРОДУКТИВНІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ГІБРИДНА АРХІТЕКТУРА, GRAPHQL, FAAS, БЕЗСЕРВЕРНІ ОБЧИСЛЕННЯ.*

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ 9

ВСТУП 10

1. ПОБУДОВА ВЕБ-СЕРВЕРУ 12

1.1. ЗАВДАННЯ ДОСЛІДЖЕННЯ 12

1.2. API ВЕБ-СЕРВЕРУ НА ОСНОВІ АРХІТЕКТУРНОГО СТИЛЮ REST ТА МОВИ ЗАПИТІВ GRAPHQL 20

1.3. ОСНОВНІ ПРОВАЙДЕРИ SERVERLESS 26

ВИСНОВКИ ДО РОЗДІЛУ 1 30

2. SERVERLESS - АРХІТЕКТУРА ВЕБ-СЕРВЕРУ 32

2.1. МОДЕЛЬ ВЕБ-СЕРВЕРУ 32

2.1.1. КОМПОНЕНТИ АРХІТЕКТУРИ 32

2.1.2. ЦИКЛ РОЗРОБКИ 37

2.2. РЕАЛІЗАЦІЯ GOOGLE CLOUD FUNCTIONS 39

2.3. ПЕРЕВАГИ ТА НЕДОЛІКИ ПОТОЧНОЇ РЕАЛІЗАЦІЇ SERVERLESS 45

ВИСНОВКИ ДО РОЗДІЛУ 2 49

3. РЕАЛІЗАЦІЯ ГІБРИДНОГО СЕРВЕРУ 50

3.1. ГІБРИДНА АРХІТЕКТУРА СЕРВЕРУ 50

3.2. ПОБУДОВА ДОДАТКУ-ПРОТОТИПУ 54

3.2.1. БІЗНЕС ЛОГІКА ДОДАТКУ 54

3.2.2. БАЗА ДАНИХ 56

3.2.3. СТРУКТУРА ПРОЕКТУ 58

3.2.4. БІБЛІОТЕКИ ТА ЗАЛЕЖНОСТІ 61

3.2.5. GRAPHQL API 64

3.3. КОМПІЛЯЦІЯ ТА РОЗГОРТАННЯ ЗАСТОСУНКУ 66

3.4. МОНІТОРИНГ ТА СУПРОВІД ДОДАТКУ 69

ВИСНОВКИ ДО РОЗДІЛУ 3 71

4. ТЕСТУВАННЯ ПРОДУКТИВНОСТІ ТА ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ ПРОТОТИПУ ДОДАТКУ НА SERVERLESS ТА GRAPHQL 72

4.1. ВИЗНАЧЕННЯ ПРОДУКТИВНОСТІ ТА ЕФЕКТИВНОСТІ ВЕБ СЕРВЕРУ 72

4.2. КОНФІГУРАЦІЯ ТЕСТІВ ТА ОГЛЯД СЦЕНАРІЇВ 75

4.3. СЦЕНАРІЙ А 78

4.4. СЦЕНАРІЙ Б 80

4.5. ОЦІНКА ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНИХ ПІДХОДІВ 83

4.6. РЕКОМЕНДАЦІЇ ЩОДО ЗАСТОСУВАННЯ SERVERLESS ДЛЯ ПОБУДОВИ ВЕБ-СЕРВЕРУ 85

ВИСНОВКИ ДО РОЗДІЛУ 4 86

5. СТАРТАП-ПРОЕКТ 88

ВИСНОВКИ ДО РОЗДІЛУ 5 94

ВИСНОВКИ 95

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 98

ДОДАТОК А 103

ДОДАТОК Б 105

ДОДАТОК В 106

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

Serverless – безсерверні обчислення, модель хмарних обчислень для яких платформа динамічно керує виділенням машинних ресурсів.

FaaS (англ. Function as a Service) – у даному дослідженні синонім Serverless.

PaaS (англ. Platform as a service) – модель надання хмарних обчислень, при якій споживач отримує доступ до використання інформаційно-технологічних платформ: операційних систем, систем управління базами даних, зв'язного програмного забезпечення, засобів розробки і тестування розміщених у хмарних провайдерах.

BaaS (англ. Backend as a service) – модель для надання розробникам веб-додатків і мобільних додатків способу пов’язати свої додатки з резервним хмарним сховищем та API, а також надає такі функції, як управління користувачами, push-сповіщення та інтеграція із службами соціальних мереж.

API (англ. Application Programming Interface) – набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

GraphQL – це мова запитів з відкритим вихідним кодом для API, а також середовище виконання запитів щодо наявних даних.

REST (англ. Representational State Transfer) – підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів.

Kubernetes – відкрита система автоматичного розгортання, масштабування та управління застосунками у контейнерах.

ВСТУП

Зростання Інтернет обчислень призвело до зростання обчислювальних центрів, внаслідок чого корпоративні центри обробки даних, як правило, складаються з великої кількості серверів, які не використовуються в повній мірі. Це в свою чергу стало причиною збільшення вартості обслуговування відповідних сервісів, що обумовлено прямою залежністю між кількістю серверів та витратами на їх оренду, електроенергію, ліцензії на програмне забезпечення та адміністрування. Саме тому на сьогодні зростає кількість корпорацій, які використовують пули серверних ресурсів, що стало можливим завдяки розвитку віртуалізації серверів, мереж та пристроїв збереження інформації. Серверний пул являє собою групу серверів в кластері, які об’єднані в деяке логічне одиницю(пул).

Особливістю інформаційних систем із клієнт-серверною архітектурою, що функціонують в сучасних мережах, є генерація великої кількості повідомлень, що передаються мережами. Довільний запит користувача на виконання конкретної задачі веб-сервісом може викликати створення та надсилання десятків запитів та відповідей до інших веб-сервісів. Одним із підходів до підвищення їх продуктивності є збільшення кількості веб-серверів, які опрацьовують запити, згенеровані користувачами, тобто клієнтами, до цього веб-сервісу. [3]

Клієнт повинен бачити веб-сервіс, до якого звертається, як єдине ціле. Для цього на декількох веб-серверах встановлюється ідентичне програмне забезпечення, а маршрутизатор надсилає запит від клієнта до одного із серверів, який вибирають за певними ознаками. Розподіл запитів між веб-серверами допомагає уникнути ситуації, коли потік запитів викликає таке навантаження на окремий сервер, що можлива швидкість обробки запитів та наявні системні ресурси потребуватимуть побудови черги. [3]

Для вирішення згаданих проблем з дотриманням необхідних вимог може бути запроваджено використання технології Serverless для розробки веб-серверу. Ключові характеристики даної інфраструктури забезпечують застосунок необмеженою кількістю обчислювальних ресурсів, що залучаються лише поли вони потрібні.

До 2015 року завдання створення хорошого API майже завжди призводило до того, що розробники вибирали REST незалежно від фактичного напрямку використання. Fielding визначає REST [5] як архітектурний стиль навколо моделі мислення, що використовується при створенні стандартів, що описують Інтернет: протокол передачі гіпертексту (HTTP), уніфікований ідентифікатор ресурсу (URI) та мова розмітки HyperText (HTML) [6]

У 2015 році Facebook відкрив специфікацію технології, що називається GraphQL. Це підхід, який визначає декларативну мову запитів для отримання даних з API. Він продемонстрував, що є випадки, коли для задоволення вимог системи необхідно щось інше, ніж HTTP. Наштовхуючись на два підходи до створення API, виникає питання, який вибрати для конкретного випадку? Впровадження систем коштує дорого. Тому дуже важливо заздалегідь правильно визначити, який підхід вибрати для конкретної задачі. Однією із задач даної дипломної роботи полягає у здійсненні порівняння між технологією GraphQL та принципами та обмеженнями, зазначеними у архітектурному стилі REST, з метою оцінки їх ключових відмінностей. На основі даного аналізу здійснено використання GraphQL для побудови додатку прототипу. Згадані розбіжності повинні допомогти розробникам у прийнятті обґрунтованого рішення, відповідно до завдання вибору найкращого підходу для конкретного випадку.

Отже, розробка архітектури веб-серверу потребує значних зусиль, спрямованих на забезпечення гнучкої інфраструктури середовища розгортання серверу. Крім цього, потрібно врахувати, що значна частина виділених та придбаних обчислювальних ресурсів не використовується протягом більшої частини часу існування серверу. Зазначені причини зумовлюють необхідність дослідження можливості застосування безсерверних обчислень (Serverless) та мови запитів GraphQL для побудови веб-серверу.

1. ПОБУДОВА ВЕБ-СЕРВЕРУ

1.1. ЗАВДАННЯ ДОСЛІДЖЕННЯ

Необхідно надати визначення терміну архітектура програмного забезпечення. Модель Perry та Wolff [7] визначає архітектуру програмного забезпечення як сукупність архітектурних елементів, які мають певну форму у зв’язку з конкретними причинами. Далі вони класифікують ці елементи на елементи з'єднання, обробки та передачі даних. Fielding далі розширює цю модель, виключаючи конкретні причини [5]. Він обґрунтовує це тим, що, хоча вони є важливими для еволюції системи та її архітектури, після реалізації програмного засобу спосіб роботи системи не залежить від її первісного обґрунтування. Стандарт ANSI/IEEE 1471-2000 [8], згодом замінений ISO/IEC/IEEE 42010: 2011 [9], застосовує аналогічний підхід, визначаючи архітектуру програмного забезпечення як «основні концепції та властивості системи в її середовищі, виражені через її елементи , взаємозв'язки та принципи її проектування та еволюції»[9]. Цікаво, що це визначення згадує принципи проектування як частину архітектури, хоча Fielding вже обґрунтовано стверджував, що призначення додатку не може бути частиною архітектури програмного забезпечення. Натхненний моделлю Perry та Wolff, Fielding визначав архітектуру програмного забезпечення як "абстракцію елементів виконання програмної системи протягом певного етапу її роботи" [5]. Це визначення підкреслює два моменти:

1. Архітектура - це абстракція елементів середовища виконання програми.

2. Вона може змінюватися залежно від фази роботи програми. Fielding згадує приклад файлу конфігурації [5], який вважається елементом даних під час фази запуску, але втрачає атрибут архітектурного елемента в подальшому.

Clements та ін. [10] та Bass [11] згодом спираються на це визначення та чітко визначають компоненти як сутність що існує під час виконання програми, тим самим надалі демонструючи важливе значення середовища виконання як частини архітектури програмного забезпечення. Bass та ін. дійшли висновку, що «архітектура програмного забезпечення системи - це сукупність структур, необхідних для розуміння системи, які містять програмні елементи, відносини між ними та властивості обох» [11]. Сукупність структур у їх визначенні - це сукупність елементів, які утримуються разом завдяки певному зв’язку. Вони пояснюють, що жодна окрема структура не може повністю описати архітектуру системи і тому необхідно визначати її як сукупність структур.

Вони класифікуються на три різні типи:

- Модульні структури;
- Компонентно-з'єднувальні структури;
- Структури розподілення ресурсів.

Перший тип стосується розподілу систем на модулі реалізації, тим самим стосуючись лише статичного подання програмного забезпечення. На відміну від цих модулів, компонентно-з'єднувальні структури розглядають аспект середовища виконання програмного забезпечення. Один модуль, наприклад клієнт у системі Client/Server, може привести до виникнення 10 компонентів під час виконання. Один модуль може бути використаний у кількох компонентах під час виконання, тоді як компонент, як правило, складається з декількох модулів. Третій тип, структури розподілення ресурсів, стосується відношенням компонентів системи до непрограмних структур – апаратних засобів, наприклад, центральні процесори (CPU). Такі рішення, як розділення вихідного коду на окремі модулі, також називаються структурними і тому є частиною архітектури програмного забезпечення. [4]

Ще один цікавий, хоча і менш формальний погляд на архітектуру програмного забезпечення представлений Fowler. Він визначає архітектуру програмного забезпечення як "речі, які важко змінити" [12]. Klusener та ін. пізніше пропонують

подібне визначення архітектури програмного забезпечення: "ті аспекти, які найскладніше змінити" [13]. Fowler пояснює, що на відміну від архітектури будівлі, архітектура програмного забезпечення не обмежується фізикою. Це дозволяє спроектувати кожен елемент програмної системи, який можна легко змінити(замінити) – що є не тривіальною задачею в архітектурі будівлі. Однак, оскільки складність збільшується для кожного змінного елемента, врахування усіх можливих змін неможливий [12].

Підсумовуючи, термін архітектура програмного забезпечення може бути виражений абстракціями аспектів виконання програмної системи. Абстракції служать необхідності приховувати складність та деталізацію з метою спрощення. Однак абстракції завжди є компромісом між деталями, які дозволяють точний контроль та узагальненнями, що спрощують речі. Розробнику доводиться вибирати, які деталі слід приховати, щоб легше змінити згодом. Однак змінити саму абстракцію важко і дорого. Це і є обґрунтуванням визначень Fowler, Klusener та інш.

Веб-сервер – це апаратно-програмна платформа, підключена до локальної або глобальної мережі, на якій буде розміщено програмний модуль або групу модулів, які здійснюватимуть обробку клієнтських запитів, а також формуватимуть відповіді на ці запити у відповідності до логіки, що реалізована у цих модулях. Апаратну частину веб-сервера становить серверний комп’ютер, програмну – операційна система та набір службових додатків, а також набір модулів, що реалізують логіку обробки клієнтських запитів. Службові модулі – це модулі, що реалізують роботу з мережевими інтерфейсами і надають необхідне програмне оточення для роботи модулів обробки клієнтських запитів. [1]

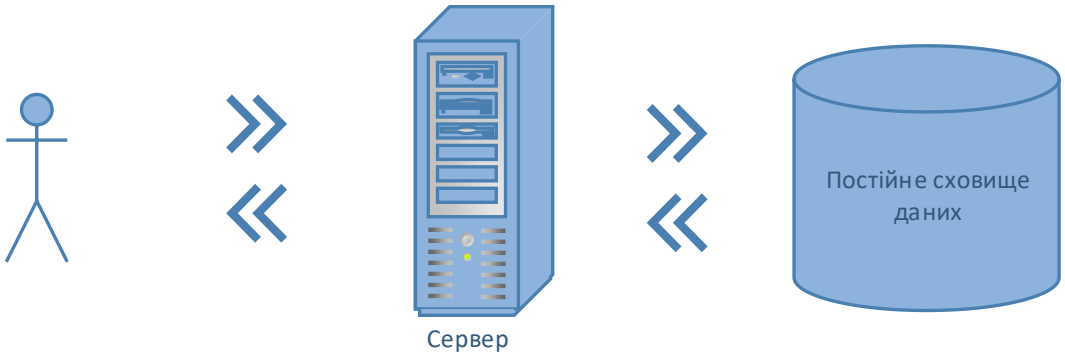


Рисунок. 1.1 Схематичне зображення взаємодії клієнту та серверу

Зазвичай, веб-сервери отримують запити через мережу по протоколу HTTP (або HTTPS у випадку використання захищеного каналу передачі) або через інші протоколи, надбудовані над ним, наприклад XML-RPC чи SOAP, тощо. Також веб-сервери можуть використовувати базу даних чи інші репозиторії для зберігання необхідних даних (рисунок 1.1). Зазвичай, база даних розташована на окремому сервері, і обмін даними з веб-сервером здійснюється через локальну чи глобальну мережу за допомогою протоколів, специфічних для систем управління базами даних. Отже, веб-сервер є достатньо складним об’єктом, коректна робота та швидкодія якого залежить від багатьох чинників. [1]

Побудова веб-серверу вимагає від розробника вирішення численних складних архітектурних завдань:

- Забезпечення надлишкової обчислювальної потужності.
- Географічне розповсюдження додаткових копій для збереження послуги у разі несправності.
- Балансування навантаження і маршрутизація для ефективного використання ресурсів.
- Масштабування вгору або вниз у відповідь на зміну навантаження системи.
- Моніторинг стану серверу.
- Ведення журналу системних повідомлень, необхідних для відладки або налаштування продуктивності.

- Оновлення системи, включаючи виправлення безпеки.
- Міграція до нових серверів.

Протягом періоду існування програмних засобів розробники намагалися полегшити вирішення зазначених завдань шляхом розвитку інфраструктури. Спочатку розробники закуповували або орендували спеціалізовані машини. Початкові капітальні витрати та поточні експлуатаційні витрати були високими. Час збільшення пропускної спроможності був довгим, а забезпечення пікових обчислювальних навантажень в системах з різним попитом вимагало попереднього планування, а часто і забезпечення багатьох запасних машин. З розвитком хмарних обчислень розробники перейшли з фізичних машин на віртуальні машини. Драматичні скорочення часу запуску призвели до можливості масштабування розгортання програми вгору та вниз у відповідь на зміни попиту, оплачуючи користування машиною погодинно. Разом з інструментами автоматичного масштабування це дозволило значно зменшити експлуатаційні витрати, але все-таки вимагало, щоб розробник чітко керував своїми віртуальними машинами. Пропозиції Platformas-a-Service (PAAS), такі як Heroku та Google App Engine, забезпечили шар абстракції поверх хмарних систем, щоб полегшити операційне навантаження, хоча за певну ціну з точки зору гнучкості та контролю (див. рисунок 1.2).

Розвиток інфраструктури спричинив до виникнення такого різновиди розгортання додатків як Serverless. Безсерверна архітектура (Serverless Architecture) реалізує спосіб побудови додатків і служб без необхідності управління серверною інфраструктурою (див. рисунок 1.3).

Створене таким чином програмне забезпечення виконується на сервері, проте все керування цим сервером виконується сторонньою організацією – провайдером.

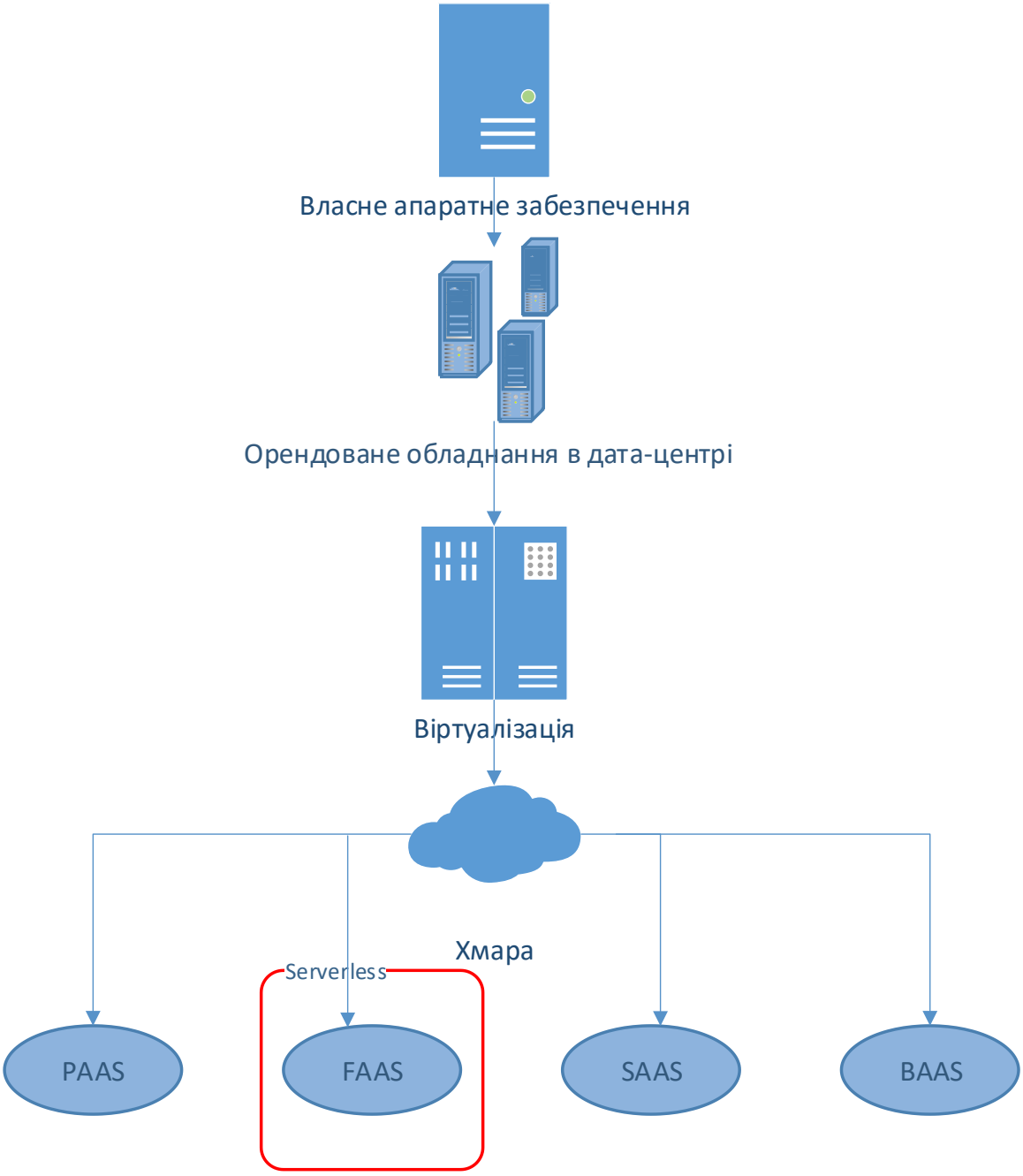


Рисунок. 1.2 Еволюція середовищ розгортання веб-серверу

Serverless є платформою що дозволяє побудувати розподілений веб-сервер. Розподілені обчислення або розподілена обробка даних – спосіб розв'язання трудомістких обчислювальних завдань з використанням двох і більше комп'ютерів, об'єднаних в мережу. Розподілені обчислення є окремим випадком паралельних

обчислень, тобто одночасного розв'язання різних частин одного обчислювального завдання декількома процесорами одного або кількох комп'ютерів. Тому необхідно, щоб завдання, що розв'язується, було сегментоване – розділене на менші задачі, що слабо пов’язані між собою та можуть обчислюватися паралельно. Особливістю розподілених обчислювальних систем, на відміну від локальних суперкомп’ютерів, є можливість майже необмеженого приросту продуктивності за рахунок горизонтального масштабування. [14]

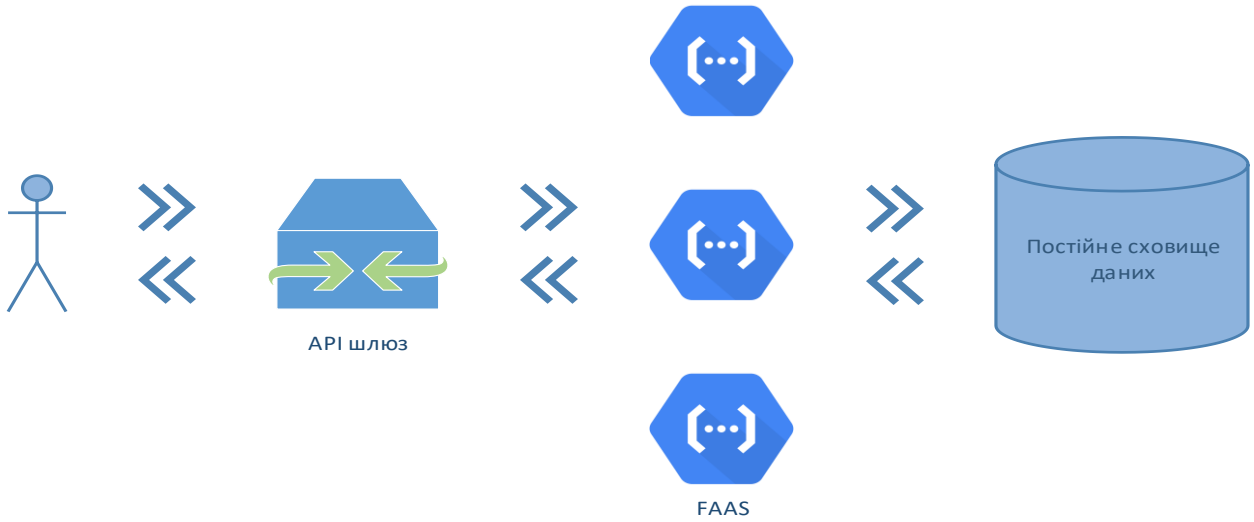


Рисунок. 1.3 Схематичне зображення Serverless архітектури

Існування Serverless вимагає дослідження можливості побудови веб-серверу, що за своїми якісними характеристиками буде дорівнювати або перевершувати існуючи традиційні рішення. У процесі розробки архітектури веб-серверу з використанням технології Serverless потрібно вирішити ряд питань пов’язаних з особливостями розгортання додатку у даному середовищі.

Таким чином, **мета дослідження** полягає в розробці архітектурного рішення для побудови веб-серверу на основі технології Serverless.

Для досягнення поставленої мети були сформульовані наступні **завдання**, що визначили логіку дослідження та його структуру:

– дослідити еволюцію підходів побудови середовищ розгортання веб-серверу;

- провести аналіз випадків можливості та доцільності застосування Serverless (безсерверних обчислень) для побудови веб-серверу;
- провести порівняльний аналіз традиційної та Serverless архітектури веб-серверу, виявити переваги та недоліки застосування безсерверних обчислень у веб-середовищі;
- розробити гібридну архітектуру веб-серверу для використання переваг Serverless та традиційної архітектур в одному застосунку;
- реалізувати API веб-серверу за допомогою GraphQL;
- провести обчислювальні експерименти з визначення продуктивності та ефективності застосування трьох підходів до розробки веб-серверу – традиційного, Serverless та гібридної архітектури;
- розробити рекомендації щодо застосування Serverless для побудови веб-серверу.

1.2. API ВЕБ-СЕРВЕРУ НА ОСНОВІ АРХІТЕКТУРНОГО СТИЛЮ REST ТА МОВИ ЗАПИТІВ GRAPHQL

Термін API використовується в різних контекстах в області інженерії програмного забезпечення. Загалом, це стосується інтерфейсу програмного елемента, який можна викликати або виконати. Зазначені програмні елементи з'являються на різних рівнях абстракції програмної системи. У даному дослідженні мова йде лише про ті елементи, які з’являються на найвищому рівні: компоненти та канали комунікації. Щоб ще більше звузити сферу застосування, ми обмежимо кількість систем, що розглядаються, на ті, які споживаються лише машинами, а не людиною. Отже, API в контексті цієї роботи є компонентом, який пропонує певні функціональні можливості та дані іншим компонентам через визначений інтерфейс, який дозволяє їм виконувати різні завдання, але не пропонує користувальницького інтерфейсу, який безпосередньо представлений користувачеві.

GraphQL - це технологія, що створена в Facebook у 2012 році [16]. Проект специфікацій [17] був відкритий у 2015 році. GraphQL описується як "мова запитів для вашого API" [18] і була побудований у відповідь на проблеми з продуктивністю під час переходу Facebook на нативні мобільні додатки [16]. Це технологія побудови API була розроблена як альтернатива REST. Незважаючи на свою назву, вона не лише дозволяє запитувати дані з сервера, але і змінювати їх. Далі ми розглянемо, як працює GraphQL.

Схема

Основою для GraphQL є схема. Вона описує існуючі типи та їх відносини, а також точки входу для клієнтів – запити та мутації. Рисунок 1.4 показує зразкову схему GraphQL. У корені Schema визначається елемент схеми, який далі поділяється на запит та мутацію. Конкретний тип «User» має ряд полів. Знак оклику вказує на урегульованість, тобто поле не може бути нульовим. Квадратні дужки навколо

визначення типу вказують на масив даного типу. Параметри, як у запитах так і у мутаціях, названі, тому їх можна вказати в будь-якому порядку.

```
schema {
  query: Query
  mutation: Mutation
}

type User {
  id: ID!
  nickName: String
  posts: [Post]!
  followees: [User]!
  followers: [User]!
}

type Post {
  id: ID!
  author: User
  content: String
  # The ISO representation of the date when the post was created.
  createdAt: String
  replies: [Post]!
  replyTo: Post
}

type Query {
  # Retrieve the timeline of a user identified by their nickname.
  timeline(of: String): [Post]!
  user(nickName: String): User
  users: [User]!
}

type Mutation {
  writePost(authorNick: String, content: String, replyTo: String = null): Post
  newUser(nickName: String): User
  followUser(me: String, other: String): User
}
```

Рисунок 1.4 Визначення схеми GraphQL, що ілюструє визначення типів та їх відношень.

Для отримання даних з серверу GraphQL до нього надсилається запит. код 1.2 показує такий запит, тоді як Рисунок 1.6 ілюструє зразок результату. Він отримує вміст

та прізвисько автора всіх публікацій на часовій шкалі користувача «MaxMustermann». Зверніть увагу, як запит починається з одного з полів типу Query: «часова шкала». У схемі визначається, що запит «часова шкала» приймає єдиний параметр з ім'ям і повертає масив повідомлень. Рисунок 1.5 також ілюструє використання фрагментів – використовуючи набір полів, з описом бажаних полів кожної публікації на шкалі 8, щоб також отримати ці поля для кожного повідомлення. Застосування фрагмента аналогічно оператору Spread в ECMAScript 2015 [19]. При видачі запиту GraphQL вимагає, щоб усі запитувані поля запиту були примітивними типами з метою поліпшення передбачуваності [16].

```
query {
  timeline(of: "MaxMustermann") {
    ...basicPostFields
    replies {
      ...basicPostFields
    }
  }
}

fragment basicPostFields on Post {
  content
  author {
    nickName
  }
}
```

Рисунок 1.5 Запит GraphQL щодо схеми, визначеної в Рисуноку 1.1.

```
{
  "data": {
    "timeline": [
      {
        "content": "My first blogpost!",
        "author": {
          "nickName": "JohnDoe"
        },
        "replies": [
          {
            "content": "Super cool!",
            "author": {
              "nickName": "MaxMustermann"
            }
          }
        ]
      }
    ]
  }
}
```


Рисунок 1.6. Приклад результату виконання запиту, наведеного у Рисуноку 1.5.

Мутації

Мутації схожі на запити, за винятком того, що їм дозволяється викликати побічні ефекти, хоча це не є обов'язковим з точки зору специфікації GraphQL [18], а є лише умовою. Тому мутації використовуються для зміни даних на сервері. Рисунок 1.7. показує зразок мутації. Мутації повертають значення, як і запити. Отже, клієнт може запитувати дані на основі зворотного значення мутації.

```
mutation {
  writePost(authorNick: "JohnDoe", content: "Blogging all day long!") {
    author {
      posts {
        content
      }
    }
  }
}
```

Рисунок 1.7. Мутація GraphQL на основі схеми, визначеної в Рисуноку 1.4, для додавання нової публікації в блозі для користувача JohnDoe


```
{
  "data": {
    "writePost": {
      "author": {
        "posts": [
          {
            "content": "My first blogpost!"
          },
          {
            "content": "Blogging all day long!"
          }
        ]
      }
    }
  }
}
```


Рисунок 1.8. Результат мутації, проілюстрованої у Рисуноку 1.7

Транспортний протокол

GraphQL сам по собі є лише специфікацією для "середовища виконання запитів" [18]. Тому вона не стосується транспортного протоколу, який використовується між клієнтом і сервером для передачі цих запитів і мутацій. Оскільки GraphQL передає всю необхідну інформацію в самому тілі запиту, він також працює з іншими транспортними протоколами, такими як TCP або UDP. При використанні з HTTP зазвичай створюється єдина кінцева точка, яка приймає клієнтські запити POST, передаючи корисне навантаження в тілі HTTP.

Порівняння GraphQL з REST

Операція - це найменша одиниця роботи, яку клієнтський компонент може виконати або запитати від серверного компонента. Викликання операції призводить до взаємодії двох компонентів. Однак в одній взаємодії потенційно можна викликати кілька

операцій. Тому операцію з іменем не слід змішувати з викликом процедури, як у стилі віддаленого виклику процедури (RPC). Наприклад, операція в контексті API REST може викликати метод GET ресурсу. Однак для HTTP API в цілому деталізація не обов'язково визначається методом HTTP. Системи, що використовують HTTP виключно для тунельних цілей, можуть визначати інтерфейс, який дозволяє викликати кілька операцій в одному запиті HTTP, таким чином підтримуючи операції масового використання. Отже, виклик операції завжди призводить до взаємодії двох компонентів, але декілька операцій можна об'єднати в одну взаємодію. Взаємодії описують необхідну комунікацію в розподілених системах між компонентами лише в тому випадку, якщо вони хочуть співпрацювати у досягненні певної мети. Важливо врахувати, що можливість викликати кілька операцій в рамках однієї взаємодії повністю залежить від конструкції системи. Наприклад, під час роботи з HTTP-протоколом на рівні додатків, розробники, як правило, намагаються однозначно поєднувати операції та взаємодії, оскільки HTTP достатньо багатий, щоб описати семантику операції на рівні протоколу. GraphQL з іншого боку дозволяє виконувати кілька мутацій в одній взаємодії. [4]

Найбільшим наслідком використання GraphQL для проектування API є перенесення ряду обов'язків з сервера на клієнта. По-перше, клієнт відповідає за розробку робочого процесу, який повинен бути представлений у додатку - клієнт відповідає за реалізацію фактичних правил, які визначають, який із доступних шляхів є дійсним. Переважно через відсутність метаданих у відповідях із сервера, клієнт повинен самостійно розібратися, які кроки чи послідовності операцій є дійсними. [4]

По-друге, якщо клієнт використовує бібліотеку, яка додає кеш, він також повинен керувати його інвалідацією, завданням, яке не варто недооцінювати, коли додаток збільшується в розмірах. В залежності від проблемної області та вимог програми, вплив цих наслідків різний. Іноді кеші не потрібні або їх використання навіть не доцільне через вимоги надання даних у режимі реального часу. [4]

1.3. ОСНОВНІ ПРОВАЙДЕРИ SERVERLESS

На даний момент на ринку є невелика відносно невелика кількість компаній, що пропонують Serverless. Розглянемо найбільш поширені, особливо варто звернути увагу на потрібні нам елементи та їх особливості. Постачальників можна розділити на основні та другорядні групи. Основна група складається з найбільших публічних хмарних постачальників архітектури безсерверних обчислень.

AWS Lambda. Пропозиція FaaS, що належить до веб-сервісів Amazon, була представлена в 2014 році. З моменту виходу Lambda стала синонімом того, що означає Serverless, займаючи позицію провідного продукту на ринку з найширшим спектром доступних послуг. Напевно, найвідомішим прикладом прийняття загальнодоступних серверів є Netflix.

Azure Functions by Microsoft. Сервіс, запущений у 2016 році, конкурував з AWS Lambda. Azure Functions пропонує аналогічний набір послуг Amazon з акцентом на сімейство мов та інструментів Microsoft. Одним із прикладів використання функцій Azure є «https://haveibeenpwned.com/».

Google Cloud Functions (GCF). Один із чотирьох найбільших провайдерів, Google випустив своє рішення лише у 2017 році. Служба GCF раніше відставала від Azure та Lambda, але протягом 2018 року Google вдалося виправити попередні помилки, про що свідчать нотатки до випуску GCF.

Всі згадані провайдери пропонують подібні послуги, достатньо для запуску програми на керованій інфраструктурі. Вони також пропонують достатні можливості, щоб отримати всі переваги концепції FaaS, але вони можуть бути різними. Щоб визначити найкращий варіант для вас, порівняємо доступні послуги, використовуючи такі критерії:

- Моделі ціноутворення та коефіцієнти розрахунків;

- Мови програмування, що підтримуються;
- Типи тригерів функції;
- Тривалість виконання та паралельність;
- Методи розгортання;
- Методи моніторингу та ведення журналів.

Таблиця 1.1. Порівняння ціноутворення основних провайдерів Serverless

	Безкоштовний обсяг, місяць	**Ціна, GB/c**
AWS Lambda	1 млн 400,000 GB/c	$0.00001667
Microsoft Azure	1 млн 400,000 GB/c	$0.000016
Google Cloud Functions	2 млн 400,000 GB/c	$0.0000004

Підводячи підсумки, AWS Lambda пропонує середнє місце в ціноутворенні, тоді як Azure може відрізнятися за витратами, залежно від використовуваного процесора та пам'яті. Але для середовищ Windows Azure пропонує найнижчу ціну. Усі виробники пропонують аналогічні ціни, однак модель Google виглядає найдорожчою через окремі платежі за пам'ять та використання процесора.

Ще одним важливим фактором ціноутворення, який ми не повинні ігнорувати, є шлюз API. Наприклад, для AWS Lambda, шлюз API є обов'язковою умовою і необхідний для викликів HTTP та HTTPS. Однак у Google Cloud Functions для HTTP-запитів не потрібен шлюз API.

Це одна з головних переваг хмарних функцій Google: для функцій HTTP не потрібен шлюз API. Для AWS це сприяє значній кількості витрат і складності у виконанні вашої функції. За допомогою хмарних функцій Google ви отримуєте кінцеву точку HTTP і маршрутизація/мережа встановлюються для вас автоматично. Вам не потрібно платити додаткові витрати на шлюз API.

Таблиця 1.2. Порівняння функціональних параметрів основних провайдерів Serverless

Критерій	AWS Lambda	Azure Functions	GCF
Мови, що підтримуються	Node.js (JavaScript), Python, Java, C#, Go	C#, F#, Python, Java, Nodejs, Python, PHP	Node.js 6, Node.js 8, Python 3.7
Гранульований IAM	Політика IAM може бути приєднана до Lambda	RBAC підтримується в групі підписки та ресурсів. Функції знаходяться всередині.	Ще не підтримується
Persistent Storage	Повністю Serverless. S3 та DynamoDB	Змінні середовища, сховище blob	Cloud Storage, Cloud Datastore, Cloud SQL
Тип тригера	Широкий спектр AWS Services + API шлюзу	Microsoft service requests + HTTP webhooks, APIM, Function proxy and bindings	HTTP + Cloud Pub/Sub + Cloud Storage + Direct Triggers
Ведення журналу та моніторинг	Cloudwatch Logs & X-Ray	Azure Storage & App Insights	Stackdriver
Максимальний час виконання за запит, секунд	900	За замовчуванням 300 Преміум 600	За замовчуванням 60, до 540
Паралельне виконання, паралельних функцій	1000	Без обмежень (залежить від тригерів)	Немає обмеження для виконання HTTP. Для інших: 1000
Розгортання	Завантаження архіву до Lambda/S3, Serverless Framework, вбудоване редагування коду	Git, dropbox, visual studio, Kudu console, One Drive, завантаження архіву	CLI, завантаження архіву, вбудоване редагування коду, Cloud Storage
Максимум кількість функцій	Без обмежень	Без обмежень	1000 функцій на проект
Залежності	Пакети розгортання на кожну мову	package.json, npm, nuget	npm package.json, pip requirements.txt
Масштабованість та доступність	Автоматичне масштабування	Ручне або дозоване масштабування (App Service Plan) або швидке автоматичне масштабування (Consumption Plan)	Автоматичне масштабування
Оркестрація	AWS Step Functions	Azure Logic Apps + Durable functions	Ще не підтримується

На сьогоднішній день AWS Lambda, Azure Functions та Google Cloud Functions все готові до використання у серйозних проектах та є загальнодоступними. Розмірковуючи над часовою шкалою, AWS Lambda стала загальнодоступною на початку 2015 року,

Azure Functions наприкінці 2016 року, тоді як Google Cloud Functions, зовсім недавно, у липні 2018 року.

З такою великою різницею в часовій шкалі, очевидно, що AWS Lambda домінує над безсерверним ландшафтом за рахунок величезної спільноти розробників та широким спектром відомих кейсів використання. Більше того, він має розширені функції, такі як AWS Step Functions.

ВИСНОВКИ ДО РОЗДІЛУ 1

Дослідивши сучасні підходи до побудови веб-серверу, еволюцію засобів розгортання веб-застосунків, здійснивши аналіз зв’язку між архітектурним стилем REST та мовою запитів GraphQL, розглянувши особливості надання послуг Serverless головними провайдерами було виявлене наступне:

- Термін архітектура програмного забезпечення був предметом наукового інтересу багатьох дослідників. Більшість з них погодилося з тим, що архітектура представляє собою певну абстракцію щодо компонентів що утворюють структуру програми, а також зав’язків між ними.
- Побудова веб-серверу вимагає від розробника вирішення численних завдань щодо визначення правильної архітектури та відповідної інфраструктури. Для спрощення цих завдань було розроблено численні підходи до реалізації середовища розгортання веб-серверу.
- Безсерверні обчислення є одним з найновіших досягнень у даній сфері розробки програмного забезпечення. Головна ідея Serverless – спростити забезпечення інфраструктури веб-серверу зберігаючи при цьому найважливіші якісні показники застосунку.
- Порівнюючи GraphQL та REST було виявлено, що GraphQL вимагає більше відповідальності від клієнтів API. Однак, певні галузі не накладають жодних або лише декількох обмежень на операції, які є певним чином залежними один від одного, тим самим зменшуючи вплив нестачі динамічного виявлення, який би дозволив зробити висновок про недоліки GraphQL. Наприклад, API для програми, яка в основному надає кілька переглядів одного і того ж набору даних, ймовірно, буде містити безліч безпечних операцій для отримання цих даних різними способами та менше небезпечних. Чи підходить даний підхід у конкретному прикладі чи ні, завжди залежить від фактичних вимог, що унеможливлює викладення точних рекомендацій.

- Провайдери Serverless на даний момент надають високоефективні послуги, що можуть бути використані у додатках з нефункціональними вимогами. При цьому основні провайдери мають подібні якісні та вартісні характеристики послуги Serverless. AWS Lambda домінує над безсерверним ландшафтом за рахунок величезної спільноти розробників та широким спектром відомих кейсів використання

Сформульовано **наступні завдання** дослідження:

- дослідити еволюцію підходів побудови середовищ розгортання веб-серверу;
- провести аналіз випадків можливості та доцільності застосування Serverless (безсерверних обчислень) для побудови веб-серверу;
- провести порівняльний аналіз традиційної та Serverless архітектури веб-серверу, виявити переваги та недоліки застосування безсерверних обчислень у веб-середовищі;
- розробити гібридну архітектуру веб-серверу для використання переваг Serverless та традиційної архітектур в одному застосунку;
- реалізувати API веб-серверу за допомогою GraphQL;
- провести обчислювальні експерименти з визначення продуктивності та ефективності застосування трьох підходів до розробки веб-серверу – традиційного, Serverless та гібридної архітектури;
- розробити рекомендації щодо застосування Serverless для побудови веб-серверу.

2. SERVERLESS - АРХІТЕКТУРА ВЕБ-СЕРВЕРУ

2.1. МОДЕЛЬ ВЕБ-СЕРВЕРУ

2.1.1. КОМПОНЕНТИ АРХІТЕКТУРИ

Деякі стверджують, що безсерверні обчислення - це лише ребрендинг попередніх рішень, можливо, скромне узагальнення платформи як сервісу (PaaS) хмарних продуктів, таких як Heroku, Firebase або Parse. Інші можуть відзначити, що спільне використання веб-хостингу, популярне в 90-х роках, забезпечувало багато того, що можуть запропонувати безсерверні обчислення. Наприклад, у них була модель програмування, що дозволяла забезпечити високий рівень багатокористувальної роботи, еластичну реакцію на змінне навантаження та API стандартизованого виклику функцій, загальний інтерфейс шлюзу (CGI) [21], який навіть дозволяв прямо розгорнути вихідний код написаний мовами високого рівня, такими як Perl або PHP. Оригінальне рішення App Engine від Google, яке значною мірою було не оцінено ринком лише за кілька років до того, як серверні обчислення набули популярності, також дозволяли розробникам розгорнути код, залишаючи більшість аспектів операцій хмарному провайдеру.

Однак, інші дослідники вважають, що Serverless є значним нововведенням порівняно з PaaS та іншими попередніми моделями. Сьогодні Serverless з хмарними функціями відрізняються від своїх попередників кількома істотними ознаками: кращим автоматичним масштабуванням, сильнішою ізоляцією, гнучкістю платформи та екосистеми підтримки обслуговування. Особливо разючі зміни у автоматичному масштабуванні вперше були запропоновані AWS Lambda. Даний сервіс відслідковував навантаження з набагато більшою достовірністю, ніж серверні методи автоматичного масштабування, швидко реагуючи на зміни навантаження, коли це потрібно, і зменшуючи ресурси до нульового обсягу та нульової вартості за відсутності попиту.

Вартість нараховувалася за допомогою набагато більш простого способу, забезпечуючи мінімальну одиницю виміру у 100 мс у той час, коли інші послуги автоматичного масштабування нараховують вартість за годину. Тож сервіс вимагав з клієнта оплату за час, коли його код фактично виконується, а не за для ресурсів, що були зарезервовані для виконання програми. Ця відмінність забезпечила, що постачальник хмарних технологій забезпечував стимули для ефективного розподілу ресурсів. [20]

Безсерверні обчислення покладаються на потужну ізоляцію продуктивності та безпеки, щоб зробити можливим спільне використання обладнаннями кількома орендарями. VM-подібна ізоляція є поточним стандартом для спільного використання багатофункціональної апаратури для хмарних функцій [22], але через те, що запуск VM може зайняти багато секунд, безсерверні обчислювальні служби, використовують спеціалізовані методи для прискорення створення середовищ виконання функцій. Один із підходів, відтворений у AWS Lambda, - це підтримка «теплого пулу» VM, які потрібно лише призначити орендареві, та «активного пулу» екземплярів, які вже використовувались для запуску функції раніше і підтримуються для подальших викликів [23]. Управління життєвим циклом ресурсів та упаковка контейнеру для багатьох орендарів для досягнення високого рівня використання наявних ресурсів, є ключовими технічними умовами що уможливлюють існування безсерверних обчислень.

Кілька інших відмінностей допомогли серверним обчислювачам досягти успіху. Дозволяючи користувачам використовувати власні бібліотеки, Serverless може підтримувати набагато ширший спектр програм, ніж послуги PaaS, які тісно пов'язані з конкретними кейсами використання. Обчислення без серверів працює в сучасних центрах обробки даних і працює в набагато більших масштабах, ніж у старих спільних веб-хостинг середовищах.

Хмарні функції (тобто FaaS) популяризували парадигму Serverless. Однак варто визнати, що вони частково зобов'язані своїм успіхам пропозиціям BaaS, які існували з ще від початку зародження публічних хмар, таких як AWS S3. На наш погляд, ці сервіси є доменними, високооптимізованими реалізаціями безсерверних обчислень. Хмарні

функції представляють обчислення без серверів у більш загальному вигляді. Ми підсумовуємо цю думку в Таблиці 2.1, порівнюючи інтерфейси програмування та моделі витрат для декількох послуг.

Таблиця 2.1. Приклади безсерверних обчислювальних служб та відповідних інтерфейсів програмування та моделей витрат.

Сервіс	**Інтерфейс програмування**	**Модель вартості**
Cloud Functions	Довільний код	Час виконання функції
BigQuery / Athena	SQL-подібний запит	Кількість даних за запитом
DynamoDB	puts() та gets()	За put() or get() запит + сховище
SQS	події enqueue / dequeue	за API-виклик

Поширеним питанням при обговоренні безсерверних обчислень є те, як це стосується Kubernetes - технології «контейнерної оркестрації» для розгортання мікросервісів. На відміну від обчислень без сервера, Kubernetes - це технологія, яка спрощує управління серверними обчисленнями. Kubernetes може забезпечити короткочасне обчислювальне середовище, як і Serverless, однак має набагато менше обмежень, наприклад, на апаратні ресурси, час виконання та мережевий зв’язок. Вона також дозволяє розгортати на загальнодоступній хмарі програмне забезпечення, що було розроблене для використання локально.

Serverless, з іншого боку, вводить зміну парадигми, яка дозволяє повністю вивантажувати операційні обов'язки постачальнику, а також робить можливим багаторівневе мультиплексування. Розміщені пропозиції Kubernetes, такі як Google Kubernetes Engine (GKE) та AWS Elastic Kubernetes Service (EKS), займають середнє місце в цьому континуумі: вони вивантажують оперативне управління Kubernetes, надаючи розробникам можливість налаштувати довільні контейнери. Однією з ключових відмінностей між розміщеними сервісами Kubernetes та Serverless є модель

виставлення рахунків. Перші стягують за зарезервовані ресурси, тоді як останні за тривалість виконання функції.

Kubernetes - це також ідеальне середовище для гібридних програмам, коли частина працює на локальному обладнання а частина працює у хмарі. Ми вважаємо, що такі гібридні програми мають сенс при переході до хмари. Однак у довгостроковій перспективі вважається, що економія масштабу хмар, швидша пропускна здатність мережі, збільшення хмарних сервісів та спрощення керування хмарою за допомогою серверних обчислень зменшать важливість таких гібридних додатків. [20]

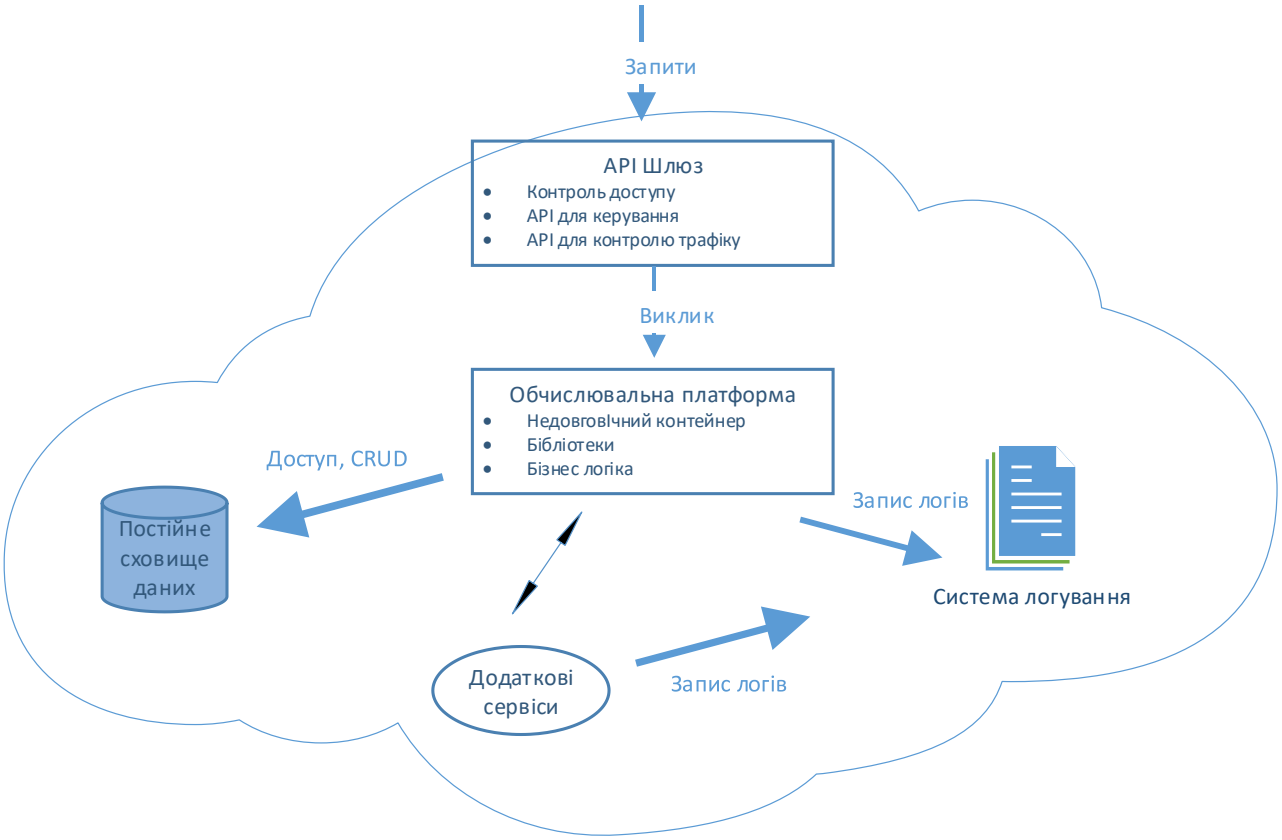


Рисунок. 2.1 Основні компоненти Serverless архітектури

Компоненти архітектури Serverless - це обчислювальна платформа Serverless, API контейнер, інструмент моніторингу роботи системи, база даних та інші доповнення. Ці компоненти - це різні послуги, що надаються та підтримуються постачальником хмарних послуг. Серед них обчислювальна платформа, що виконує код програми для обробки бізнес-логіки, включаючи доступ до бази даних та маніпуляції. Тим часом API

контейнер відповідає за управління API GraphQL з точки зору публікації всіх точок входу API, контролю доступу та контролю за трафіком. Інструмент моніторингу (або система реєстрації журналів) збирає відповідні дані та інформацію щодо роботи програми та забезпечує її доступність. І останнє, але не менш важливе, окремі сервіси, що відповідають вимогам бізнесу та можуть бути інтегровані у додаток, наприклад електронна пошта, відстеження місцезнаходження, зберігання даних та аналітика.

Наведена схема (див. рисунок 2.1) ілюструє наступні основні компоненти архітектури безсерверних обчислень. Запит до серверу здійснюється через виклик API, який спочатку переходить до контейнера API. Контейнер перевіряє, чи дозволений цей виклик, і витягує дані, що надсилаються в тілі або параметри запиту. Згодом відповідні функції в обчислювальній платформі викликаються для продовження обробки запиту. Функції опрацьовують вхідні дані на основі бізнес логіки, встановлюють, якщо це необхідно, з'єднання з базою даних для отримання або маніпулювання даними, а потім готують відповідь користувачу. Тим часом детальна інформація, що стосується процесу, відстежується та реєструється в інструменті моніторингу роботи. Таким чином, вони можуть бути проаналізовані для розслідування певного інциденту. Крім того, інструмент моніторингу повинен мати можливість відслідковувати та зберігати дані про використання будь-яких допоміжних послуг інтегрованих в веб-сервер створений на основі технології Serverless.

2.1.2. ЦИКЛ РОЗРОБКИ

Kreger [24] запропонував загальний 4-фазний життєвий цикл розробки веб-серверу. Перша фаза це побудова, в якій команда організовує всі заходи з розробки для створення компонентів веб-служби. Ці заходи складаються з планування, аналізу, проектування, впровадження та тестування. Команда розробників працює над метою випустити працездатний і перевірений сервіс для задоволення бізнес-потреб. Для цього разом із впровадженням та тестуванням приймаються до уваги різні питання, такі як визначення цілей проекту, процедур та доцільності, аналіз потреб бізнесу для виявлення вимог та очікувань, а також специфікації та дизайну послуг. [24].

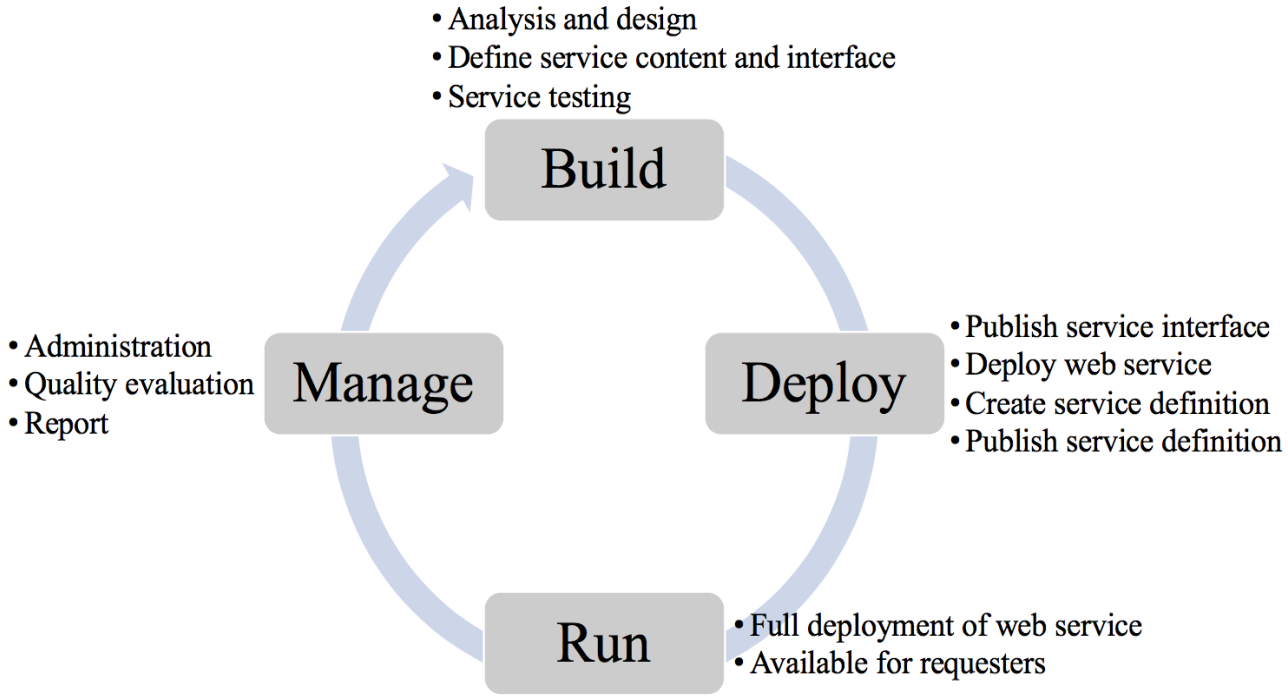


Рисунок. 2.1. Життєвий цикл розробки веб-сервісів [25] [26]

Другий етап, розгортання, стосується публікації веб-сервісу, щоб зробити його доступним споживачам. Розгортання можна класифікувати за етапами публікації інтерфейсу сервісу, розгортання веб-сервісу, побудови та визначення реалізації сервісу публікації [26]. У рамках цих кроків постачальник послуг розгортає код запуску та відповідні метадані, щоб увімкнути веб-сервіс та його доступність для користувачів послуг. Також надаються відомості про послуги, включаючи місцеположення, версію та

інструкції щодо впровадження. Крім того, план, процедура та налаштування середовища виконання також враховуються в рамках цієї фази.

По-третє, у фазі запуску веб-служба повністю розгорнута та доступна у мережі для використання. Замовник сервісу може виконувати операції «знаходження» та «виклик» [25]. Нарешті, після того, як служба функціонує, вона переходить до фази управління, яка містить адміністрування, обслуговування та оцінку програми веб-сервісу. Метою є моніторинг його ефективності в операційному середовищі, щоб визначити, чи достатньо вирішені бізнес-процеси, якість та вартість [24]. Таким чином, збираються дані та розробляються різні метрики для оцінки послуги відповідно до конкретних потреб. Наприклад, може бути записана інформація про час відгуку служби, час збільшення та зменшення пропускної здатності [24].

2.2. РЕАЛІЗАЦІЯ GOOGLE CLOUD FUNCTIONS

Середовищем виконання Serverless для даної роботи, після проведення дослідження наведеного у розділі 1.3 було обрано Google Cloud Functions. Факторами що вплинули на вибір були наступні:

- існує значна кількість досліджень спрямованих на аналіз іншого популярного рішення Amazon AWS, відповідно більший науковий сенс мав аналіз з використанням рішення, що є менш дослідженим;
- Google Cloud Functions має більший об'єм послуг що доступні на безоплатній основі, що є зручним для виконання дослідження;
- Google надає сервіси, що потрібні для використання гібридної та сучасної Kubernetes інфраструктури (див. розділ 3 та 4);
- Google надає зручний сервіс розміщення SQL БД.

Google Cloud Functions - це середовище виконання Serverless для створення та підключення хмарних служб. За допомогою хмарних функцій виникає можливість написання простих, одноцільових функції, які є пов'язаними до певних подій, ініційованих з вашої хмарної інфраструктури та сервісів. Хмарна функція спрацьовує при запуску конкретної події. Код виконується в повністю керованому середовищі. Не потрібно налаштовувати жодної інфраструктури і турбуватися про управління будь-якими серверами. [27]

Розглянемо конкретніше певні аспекти:

- код виконується в повністю керованому середовищі і не потрібно налаштовувати жодної інфраструктури. Це саме те, чого можна було очікувати від Serverless пропозиції.
- одноцільові функції: одиниця роботи, яка очікується від розробника, - це те, що ми знаємо як функції в будь-якій мові програмування. Все, що потрібно написати - це короткі фрагменти коду будь-якою мовою, яку зараз підтримує постачальник FaaS.

- пов’язаними до певних подій, ініційованих з вашої хмарної інфраструктури та сервісів. Як функція буде запускатися чи виконуватися? Завжди є спосіб безпосередньо викликати функцію, але необхідно брати до уваги випадки, коли вони викликаються побічно або асинхронно. Так, наприклад, що робити, якщо необхідно надіслати електронний лист, коли файл завантажено в Google Cloud Storage. У цьому випадку подія - це завантаження файлів у Google Cloud Storage. Підсумовуючи, ви визначаєте та налаштовуєте свою функцію, яка буде спрацьовувати при завантаженні файлів у Google Cloud Storage. Тепер, коли користувач завантажує файл у цей репозиторій, подія запускається, і оскільки функція зареєстрована для запуску, коли ця подія відбувається, Google Cloud Functions викличе функцію та передасть деталі події (файл завантажено, ідентифікатор, час, дата тощо) як параметр функції.

Google Cloud Functions – середовище виконання

Ось три ключові речі, про які слід пам’ятати, працюючи з функціями Google Cloud.

- Хмарні функції знаходяться в GA (Загальній доступності).
- Хмарні функції підтримують чимало середовищ виконання (деякі в GA, Beta та Alpha):
  - Node.js v6.14.0 (GA)
  - Node.js v8.11.1 (Beta)
  - Python 3.7.0 (Alpha)
  - Go (Альфа).

Виходячи з вищезазначених режимів виконання, повинно бути зрозуміло, що зараз можливо розробляти хмарні функції в JavaScript, Python або Go.

Як працює Google Cloud Functions?

Розглянемо схему з офіційної документації (див. рисунок 2.2.). Зліва розміщено Cloud Services. Це хмарна платформа Google та різні її сервіси, наприклад Google Cloud Storage, Google Cloud Pub/Sub, Stackdriver, Cloud Datastore тощо. Всі вони мають події, що відбуваються всередині них. Наприклад, якщо на репозиторій завантажено новий

об’єкт, видалено об’єкт або певні метадані оновлено. Аналогічно, подія відбувається якщо до логів Stackdriver або Cloud Pub/Sub додано нове твердження або отримано повідомлення, опубліковане в певній темі тощо. Не всі події підтримуються на даний момент у Google Cloud Functions.

Рисунок. 2.2. Реалізація Google Cloud Functions [27]

Коли відбувається подія, наприклад «Об'єкт завантажений у репозиторій» у хмарному сховищі, ініціюється відповідна функція, якщо вона була налаштована так, щоб її викликала ця подія. У рамках виконання функції передаються дані про подію, щоб вона міг розшифрувати те, що спричинило подію, тобто джерело події, отримати метаінформацію про подію тощо і виконати її обробку. У рамках обробки хмарна функція також може викликати інші API. Це можуть бути API Google або зовнішні API. Можливо навіть викликати іншу хмарну функцію.

Закінчивши виконувати свою логіку, Хмарна функція вказує, що вона виконана. Декілька подій призводить до відповідної кількості викликів хмарних функцій. Це все для вас вирішує інфраструктура хмарних функцій. Розробник зосереджується на логіці всередині функції та намагається зберегти спеціалізацію функції, використати мінімальний час виконання та не використати часу більше ніж це дозволено інфраструктурою. Це також говорить про те, що ця модель найкраще працює в умовах відсутності стану що необхідно зберігати між викликами. Можливо, однак, тримати стан поза Google Cloud Functions за допомогою якоїсь іншої служби, наприклад, Shared Memcache тощо.

Давайте ще раз подивимось на події, тригери та дані про події за допомогою наведеної нижче схеми (див. рисунок 2.3). Ми можемо узагальнити наступним:

- події: вони трапляються в сервісах Google Cloud Platform, наприклад, «Файл, завантажений у сховище», «повідомлення, опубліковане у черзі», «прямий запит HTTP» тощо.
- тригери: Ви можете налаштувати функції відповідати на певні тригери. Тригер - це подія та дані, пов’язані з подією.
- дані про подію: це дані, які передаються вашій хмарній функції, коли тригер події призводить до виконання вашої функції.

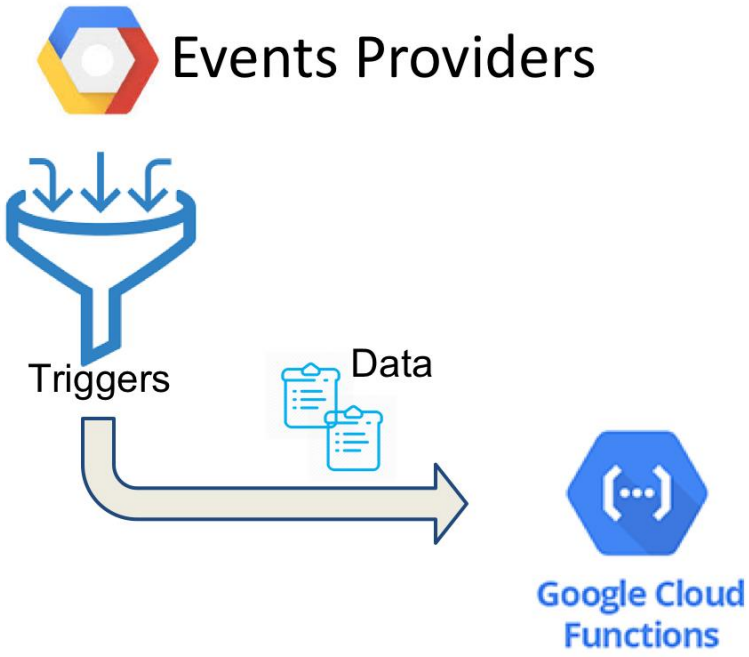


Рисунок. 2.3. Потік даних в Google Cloud Functions [27]

На даний момент Google Cloud Functions підтримує таких постачальників подій:

- HTTP - виклик функції безпосередньо через HTTP-запити
- Cloud Storage
- Cloud Pub/Sub
- Firebase (DB, Analytics, Auth)

- Логи Stackdriver
- Cloud Firestore (бета)
- Google Compute Engine (Альфа)
- BigQuery (Alpha)

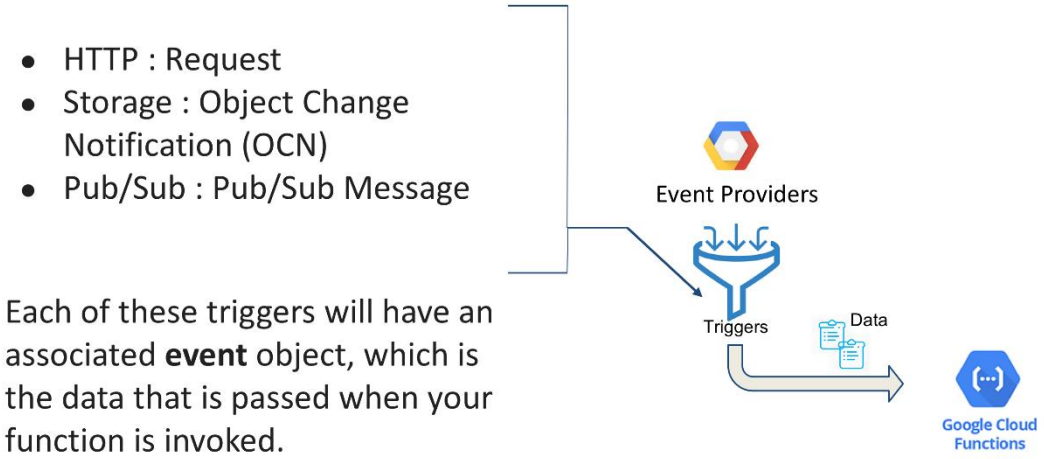


Рисунок. 2.4. Типи функцій Google Cloud Functions [27]

Провайдер надає панель керування розгорнутим сервером, де можна відслідковувати поточне навантаження, продуктивність та змінювати налаштування.

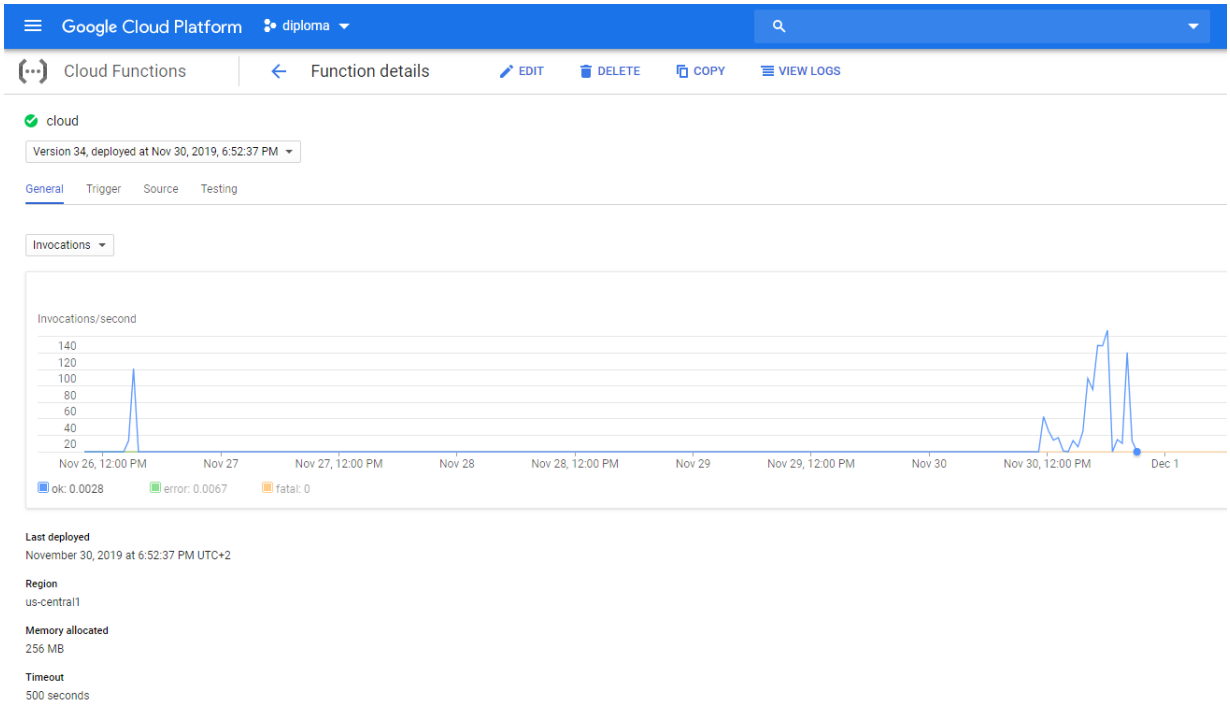


Рисунок. 2.5. Веб-інтерфейс Goggle Cloud Functions

На даний момент підтримуються два типи функцій Google Cloud (див. рисунок 3.4):

- Функції переднього плану (синхронні): Викликається безпосередньо через кінцеву точку HTTP, яка надається для вашої функції. Вони також відомі як HTTP-тригер функції.
- Фонові функції (асинхронні): вони опосередковано викликаються через подію, яка запускає вашу функцію.

2.3. ПЕРЕВАГИ ТА НЕДОЛІКИ ПОТОЧНОЇ РЕАЛІЗАЦІЇ SERVERLESS

На будь-якій Serverless платформі користувач просто записує хмарну функцію мовою високого рівня, вибирає подію, яка повинна спровокувати виконання функції, наприклад, завантаження зображення у хмарний сховище або додавання мініатюри зображень у таблицю бази даних, та дозволяє Serverless системі обробляти все інше: вибір екземпляра, масштабування, розгортання, відмовостійкість, моніторинг, ведення журналів, виправлення безпеки та інше. У таблиці 2.2 та 2.3 узагальнено відмінності між безсерверним та традиційним підходом для розробника, який у цій роботі ми будемо називати серверними хмарними обчисленнями. Зауважимо, що ці два підходи представляють кінцеві точки континууму обчислювальних платформ, орієнтованих на функції/сервери, з контейнерними рамками оркестрації, як Kubernetes, що представляють проміжні продукти.

Таблиця 2.2. Порівняння традиційної та Serverless архітектури для розробника ПЗ

Параметр	**Serverles**	**Традиційна хмара**
Коли програма виконується	При настанні певної події	Постійно
Мова програмування	JavaScript, Python. Java, Go, C#, та інш.	Будь-яка
Стан програми	Зберігається в тимчасовому сховищі (stateless)	Будь-де (stateful або stateless)
Максимальний розмір оперативної пам'яті	0.125 - 3 GiB (на розсуд Розробника)	0.5 - 1952 GiB (на розсуд Розробника)
Максимальний об'єм локальної пам'яті	0,5 GiB	0 - 3600 GiB (на розсуд Розробника)

Максимальний час виконання програми	~900 секунд	-
Мінімальна одиниця виміру	0.1 секунд	60 секунд
Вартість одиниці виміру	$0.0000002 (для 0.125 GiB оперативної пам'яті)	$0.0000867 - $0.4080000
Операційна система	На розсуд Провайдера послуг	На розсуд Розробника

Таблиця 2.3. Порівняння традиційної та Serverless архітектури для DevOps

Параметр	**Serverles**	**Традиційна хмара**
Обладнання серверу	На розсуд Провайдера послуг	На розсуд Розробника
Масштабування	Відповідальність Провайдера послуг	Відповідальність Розробника
Розгортання	Відповідальність Провайдера послуг	Відповідальність Розробника
Відмовостійкість	Відповідальність Провайдера послуг	Відповідальність Розробника
Моніторинг	Відповідальність Провайдера послуг	Відповідальність Розробника
Логування	Відповідальність Провайдера послуг	Відповідальність Розробника

У хмарному контексті серверні обчислення схожі на програмування на мові низького рівня, тоді як обчислення без сервера - це як програмування на мові вищого рівня, наприклад, Python. Програміст з Assambly, що обчислює простий вираз, такий як c = a + b, повинен вибрати один або декілька регістрів, які використовувати, завантажувати значення в ці регістри, виконувати арифметику та зберігати результат. Це відображає декілька етапів серверного хмарного програмування, де один із перших ресурсів забезпечує або ідентифікує наявні, а потім завантажує ці ресурси необхідним кодом і даними, виконує обчислення, повертає або зберігає результати та врешті-решт керує випуском ресурсів. Метою та можливістю безсерверних обчислень є надання

хмарним програмістам переваг, подібних до переходів до мов програмування високого рівня. Інші особливості середовищ програмування високого рівня також мають природні паралелі в обчисленні без серверів. Автоматизоване управління пам’яттю позбавляє програмістів від управління ресурсами пам’яті, тоді як Serverless позбавляє програмістів від управління ресурсами сервера. [20]

Підсумовуючи, головними відмінностями між бессерверними і традиційними обчисленнями є:

- слабко зв'язані обчислення та зберігання. Масштабування зберігання та обчислення є окремим - резервується та оцінюється незалежно. Загалом, зберігання забезпечується окремою хмарною службою, а обчислення не має збереженого стану;
- виконання коду без управління ресурсами. Замість того, щоб запитувати ресурси, користувач надає фрагмент коду, а хмара автоматично надає ресурси для виконання цього коду;
- оплата здійснюється пропорційно використовуваним ресурсам замість ресурсів, що резервуються. Платежі залежать від певних параметрів виконання програми, наприклад, часом виконання. На відміну від потужності та кількості виділених віртуальних машин.

Зазначені особливості зумовлюються ряд переваг Serverless перед традиційною хмарною інфраструктурою:

- велика масштабованість і гнучкість серверної частини додатку
- менше зусиль для підтримки інфраструктури проекту
- більша придатність до автоматичного тестування логіки програми, інкапсульованої в чисті функції
- простота експериментів і запровадження нового функціоналу
- сплачувати необхідно лише за час коли програма активно обробляє події, а не тоді, коли вона перебуває в стані очікування

- сплачувати необхідно лише за час коли програма активно обробляє події, а не тоді, коли вона перебуває в стані очікування. Це, по суті, означає, що час простою програми є безкоштовним, що знижує витрати на роботу сервера.
- заохочується, надання можливості клієнтським додаткам безпосередньо звертатися до ресурсів, які традиційно вважаються «backend». Дає можливість видалити дорогі компоненти, які традиційно були потрібні для виконання функції авторизації та автентифікації.
- написання фрагментів програми як окремих незалежних частин логіки дозволяє використовувати спеціалізовані послуги 3-х сторін для виконання різних завдань, що призводить до загального економічного ефекту.

У той же час існує ряд обмежень платформи Serverless, що необхідно враховувати при розробці веб-серверу:

- повністю Serverless додаток не підходить для програм у реальному часі, які використовують WebSockets, або подібні технології, оскільки функції FaaS мають обмежений термін служби;
- через деякий час бездіяльності функція повинна буде пройти “холодний” старт, який може зайняти до декількох секунд;
- залежність від постачальника. Різні постачальники послуг FaaS можуть відрізнятися в деяких особливостях використання своїх послуг, що перешкоджає переходу на іншого постачальника;
- відсутність міжпроцесового стану;
- необхідно адаптувати архітектуру додатків і процеси розвитку бізнесу;
- складність інтеграційного тестування.

ВИСНОВКИ ДО РОЗДІЛУ 2

У результаті дослідження Serverless як архітектури веб-серверу було виявлено наступне:

1. Serverless є значним нововведенням порівняно з PaaS та іншими попередніми моделями. На даний момент Serverless з хмарними функціями відрізняються від своїх попередників кількома істотними ознаками: кращим автоматичним масштабуванням, сильнішою ізоляцією, гнучкістю платформи та екосистеми підтримки обслуговування.

2. Компоненти архітектури Serverless - це обчислювальна платформа Serverless, API контейнер, інструмент моніторингу роботи системи, база даних та інші доповнення. Ці компоненти - це різні послуги, що надаються та підтримуються постачальником хмарних послуг. Серед них обчислювальна платформа, що виконує код програми для обробки бізнес-логіки, включаючи доступ до бази даних та маніпуляції.

3. Розробка веб-серверу передбачає існування життєвого циклу – побудова, розгортання, робота програми та управління.

4. Для реалізації Serverless архітектури у даному дослідженні було обрано Google Cloud Function оскільки даний провайдер має нижчу вартість та більш широкий спектр супутніх послуг, що також були застосовані при розробці додатку прототипу (див. розділ 3.2).

5. Було здійснено аналіз спільних та відмінних характеристик традиційного та Serverless середовищ розгортання веб-серверу. Головними відмінностями між бессерверними і традиційними обчисленнями є слабко зв'язані обчислення та зберігання, виконання коду без управління ресурсами оплата здійснюється пропорційно використовуваним ресурсам замість ресурсів, що резервуються.

3. РЕАЛІЗАЦІЯ ГІБРИДНОГО СЕРВЕРУ

3.1. ГІБРИДНА АРХІТЕКТУРА СЕРВЕРУ

Розглянемо окремі недоліки традиційної інфраструктури, що зумовлюють інтерес до можливості заміни її Serverless. На рисунку 3.1 зображено проблему необхідності розміщення додаткових ресурсів для обробки пікових навантажень традиційною інфраструктурою. Незважаючи на те, що переважну частину часу навантаження серверу є низьким, для обробки пікових навантажень необхідно тримати, а отже і оплачувати, значні обчислювальні ресурси.

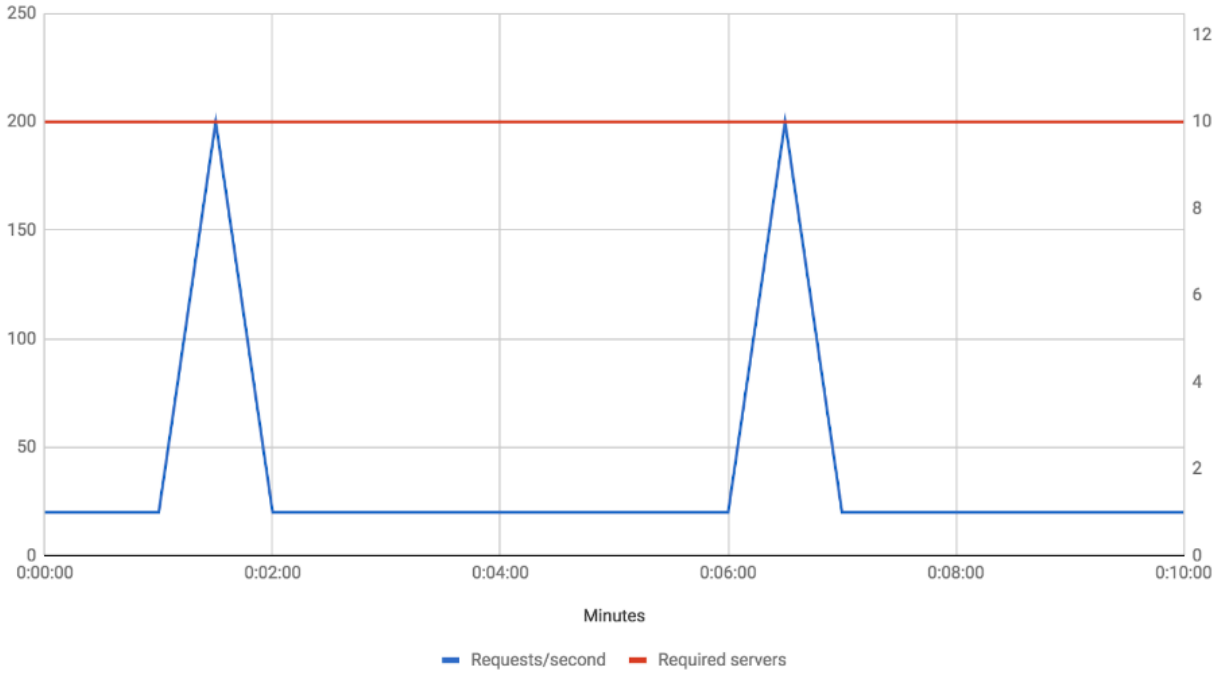


Рисунок. 3.1. Необхідність розміщувати додаткові ресурси для обробки пікових навантажень традиційною інфраструктурою

Надлишкові обчислювальні потужності також мають місце коли йде відносно поступовий приріст навантаження. (див. рисунок 3.2). Зі збільшенням попиту, розробники повинні забезпечувати ще більше надлишкових потужностей, що відповідно збільшує витрати на інфраструктуру.

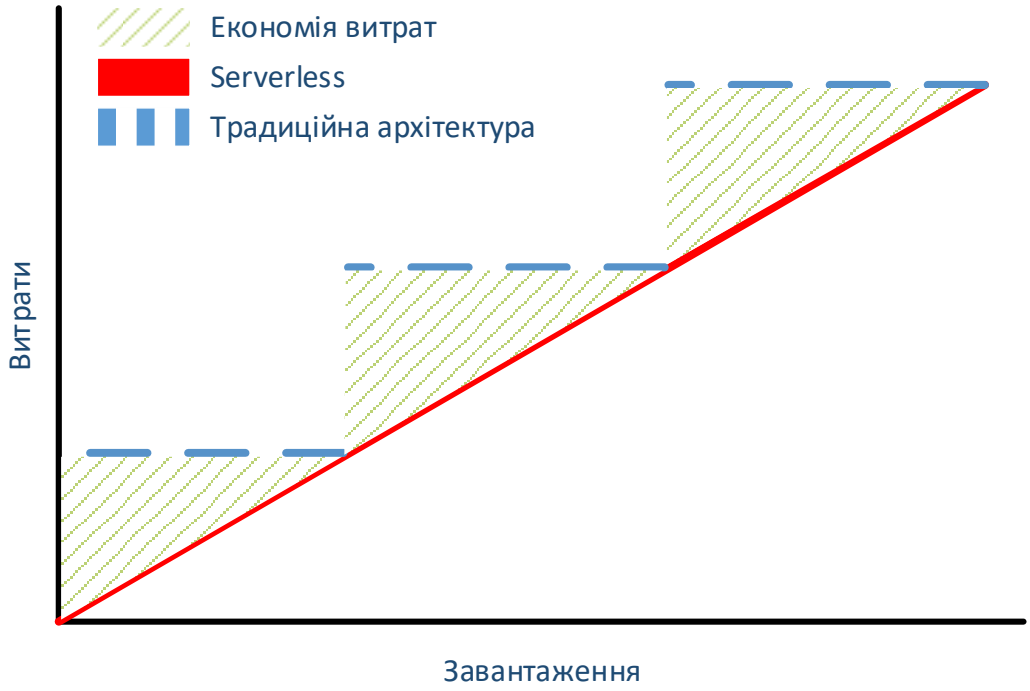


Рисунок. 3.2. Економія витрат за рахунок впровадження технології Serverless

Враховуючи вищезазначене, та наведені у розділі 2.3 переваги та недоліки технології Serverless для розробки веб-серверу, було запропоновано поєднати традиційну та Serverless архітектуру з метою нівелювання недоліків та збереження переваг обох підходів (див. таблицю 3.1).

Таблиця 3.1. Гіпотеза про необхідність розробки Гібридної архітектури

Параметр	**Традиційна**	**Serverless**	**Гібридна**
Необхідність управління інфраструктурою	Так	Ні	Частково
Зберігання стану між викликами	Так	Ні	Так
Підключення WebSocket	Так	Ні	Так
Оплата за використані ресурси	Ні	Так	Частково
Автоматичне масштабування	Ні	Так	Так
Автоматичне забезпечення пікових навантажень	Ні	Так	Так
Необхідність підтримки однієї інфраструктури	Так	Так	Ні
Спрощена процедура розробки	Так	Ні	Так

Гібридна архітектура – це поєднання традиційної хмарної інфраструктури з Serverless з метою вирішення таких завдань як необхідність зберігання стану тривалий час (наприклад WebSocket), необхідність роботи з традиційними базами даних (обмежені можливості щодо масштабування), поступова міграція до Serverless інфраструктури, підвищені вимоги до продуктивності, необхідність опрацьовувати непередбачувані пікові навантаження.

Гібридна архітектура заснована на традиційній, однак, за необхідності, передбачає можливість автоматичного залучення додаткових обчислювальних ресурсів шляхом звернення до інфраструктури Serverless (див. рисунок 3.3)

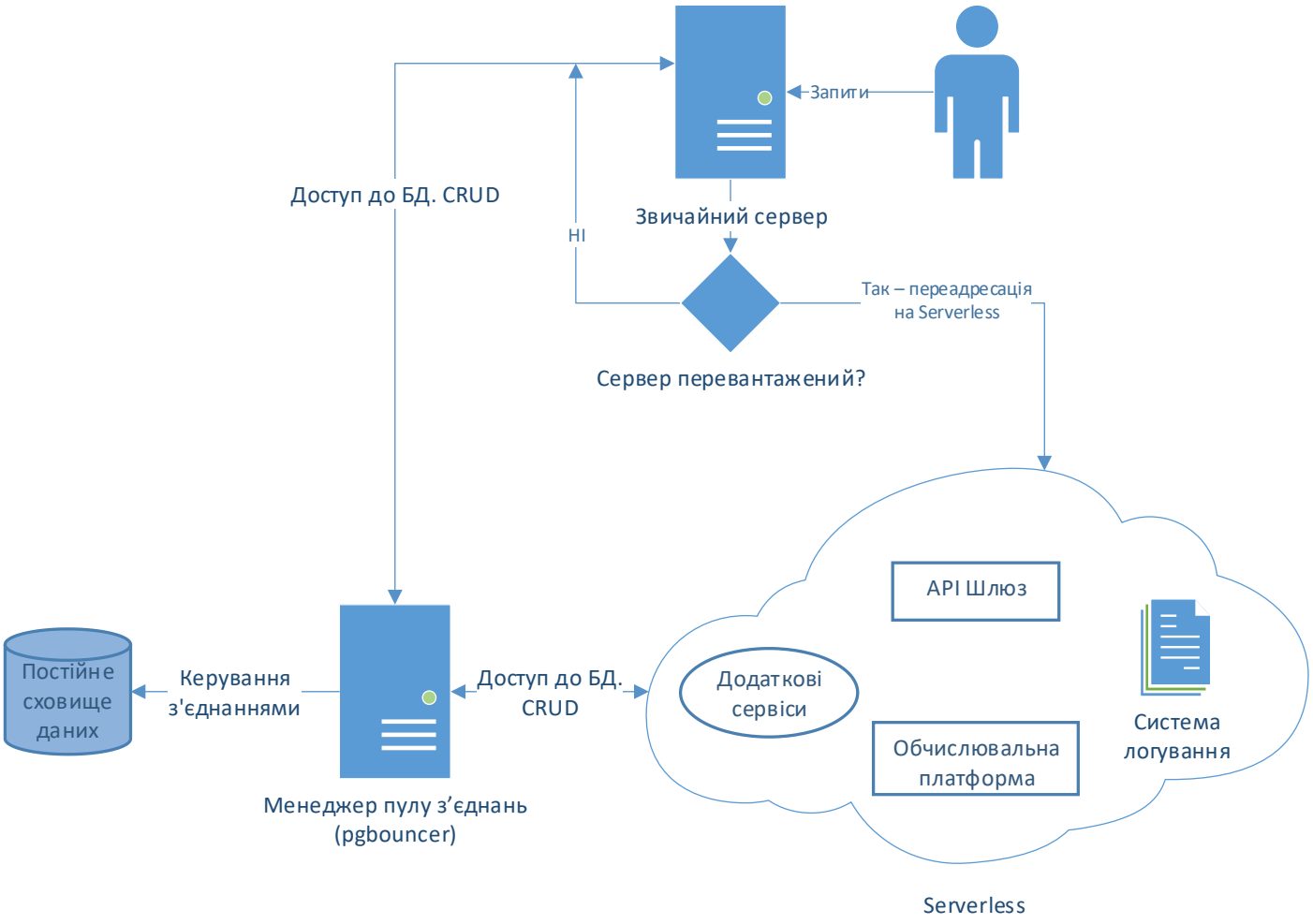


Рисунок. 3.3. Схематичне зображення гібридної архітектури

Розглянемо окремі складові гібридної архітектури:

- Звичайний сервер – точка доступу до програмної системи. Запити клієнта надходять на сервер де відбувається перевірка поточного стану завантаження системи. Якщо система перебуває під навантаженням, запит переадресовується на інфраструктуру Serverless.
- Serverless – безсерверне середовище розгортання застосунку. Пристосоване до обробки нерівномірного навантаження.
- Менеджер пулу з’єднань – сервер, що має низькі вимоги до продуктивності апаратного забезпечення і єдиним завданням якого є розпоряджання пулом з’єднань до сервісу постійного зберігання даних (бази даних). Необхідний для забезпечення стійкого зв’язку між обчислювальними потужностями та базою даних, враховуючи обмежену кількість підключень до бази даних, що може бути встановлена одночасно.
- Постійне сховище даних – база даних. Потенційно може бути місцем, що обмежує продуктивність системи, оскільки не масштабується на рівні з Serverless.

3.2. ПОБУДОВА ДОДАТКУ-ПРОТОТИПУ

3.2.1. БІЗНЕС ЛОГІКА ДОДАТКУ

Для аналізу характеристик розробленої архітектури було створено додаток-прототип. Застосунок реалізує систему обслуговування процесу надання послуг клієнтам для певного бізнесу і надає можливість забронювати отримання певної послуги у певного спеціаліста у певній локації. У рамках даного дослідження головний інтерес представляє API додатку, оскільки саме воно представляє інтерес з точки зору оцінки веб-серверу.

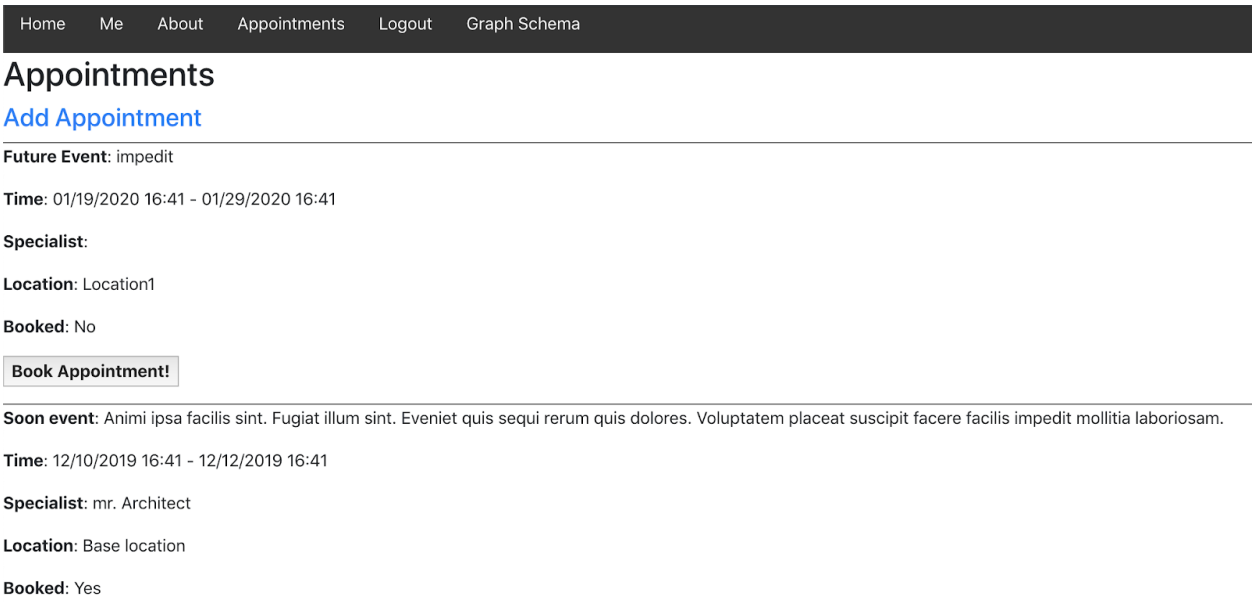


Рисунок. 3.4. Приклад веб-інтерфейсу додатку – список доступних та недоступних записів на отримання послуги

Додаток має наступні скаладові:

- Технічний стек: TypeScript, Node.js, React.js, Jest
- GraphQL для побудови API
- Реляційна база даних PostgreSQL
- Юніт та інтеграційне тестування за допомогою in-memory SQL DB

- Google Cloud Functions для Serverless
- Google Kubernetes Engine для звичайної та гібридної архітектури
- blazemeter.com, autocannon, Jmeter для тестування продуктивності
- GraphQL Voyager інструмент репрезентації GraphQL API, як інтерактивного графу

3.2.2. БАЗА ДАНИХ

Для розробки додатку-прототипу було використано базу даних PostgreSQL. PostgreSQL - це об'єктно-реляційна система управління базами даних (ОРСУБД, ORDBMS), заснована на POSTGRES, Version 4.2 - програмі, розробленою на факультеті комп'ютерних наук Каліфорнійського університету в Берклі. У POSTGRES з'явилося безліч нововведень, які були реалізовані в деяких комерційних СУБД набагато пізніше.

PostgreSQL - СУБД з відкритим вихідним кодом, основою якого був код, написаний в Берклі. Вона підтримує більшу частину стандарту SQL і пропонує безліч сучасних функцій:

- складні запити
- зовнішні ключі
- тригери
- змінювані уявлення
- транзакційна цілісність
- багатоверсійність

Крім того, користувачі можуть всіляко розширювати можливості PostgreSQL, наприклад створюючи свої:

- типи даних
- функції
- оператори
- агрегатні функції
- методи індексування
- процедурні мови

А завдяки вільній ліцензії, PostgreSQL дозволяється безкоштовно використовувати, змінювати і поширювати всім і для будь-яких цілей - особистих, комерційних чи навчальних.

Для розгортання інфраструктури бази даних було використано Google Cloud SQL. Cloud SQL - це повністю керована послуга бази даних, яка дозволяє легко налаштувати, підтримувати, керувати та адмініструвати реляційні бази даних PostgreSQL, MySQL та SQL Server у хмарі. Cloud SQL пропонує високу продуктивність, високу доступність, масштабованість та зручність. Побудована на надійній інфраструктурі, яка має приватну глобальну мережу Google та безпеку світового класу, Cloud SQL піклується про загальні завдання управління базами даних, дозволяючи розробнику зосередитись на створенні додатків.

Схема бази даних представлена наступними таблицями:

- Клієнт
- Подія
- Локація
- Спеціаліст
- Локації спеціалістів
- Профіль користувача
- Участь клієнта в подіях
- Події клієнта

Зв'язок між сутностями застосунку зручніше розглянути на основі графу API GrapgQL (див. додатки Б, В).

3.2.3. СТРУКТУРА ПРОЕКТУ

Розглянемо структуру проекту з побудови додатку-прототипу. Проект розділений на 4 модулі (див. рисунок 3.5)

- Веб-інтерфейс клієнта (не є предметом даного дослідження);
- Веб-сервер;
- Скрипти для здійснення розгортання серверу на різних середовищах (див. Розділ 3.3);
- Скрипти для здійснення тестування продуктивності веб серверу (див. Розділ 4.2).

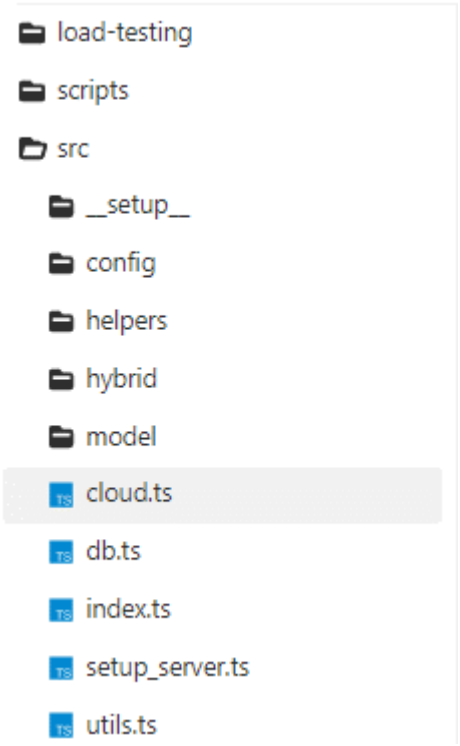


Рисунок. 3.5. Структура проекту

Сервер реалізований з використанням фреймворку TypeGraphql. Він спрямований на полегшення інтеграцій мови програмування TypeScript та GraphQL. Основна ідея - мати лише одне джерело істини, визначаючи схему за допомогою класів та декораторів. Додаткові функції, такі як ін'єкція залежностей, перевірка автентифікація та авторизація допомагають у виконанні загальних завдань.

Для роботи з базою даних використано TypeORM. TypeORM - це ORM, яка може працювати в платформах NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo та Electron і може використовуватися з TypeScript і JavaScript (ES5, ES6, ES7, ES8). Її мета - завжди підтримувати найновіші функції JavaScript та надавати додаткові функції, які допомагають розробляти будь-який додаток, який використовує бази даних - від невеликих додатків з кількома таблицями до великих масштабних корпоративних програм з декількома базами даних.

TypeORM підтримує як моделі Active Record, так і Map Mapper, на відміну від інших існуючих в даний час JavaScript ORM, що означає, що існує можливість найефективніше писати високоякісні, нещільно поєднані, масштабовані програми, що легко підтримуються. На TypeORM сильно вплинули інші ORM, такі як Hibernate, Doctrine та Entity Framework.

Відповідно, використання фреймворків дозволяє спростити розробку та мати декларативний код, який легше підтримувати (див. рисунок 3.1).

```
import { Field, ObjectType } from 'type-graphql';
import {
  Entity,
  JoinColumn,
  JoinTable,
  ManyToMany,
  OneToMany,
  OneToOne,
} from 'typeorm';

import { Event } from '../event/event-type';
import { User } from '../core/user';
import { Lazy } from '../../helpers';
import { Profile } from '../profile/profile-type';
import { ClientEventParticipation } from '../client-event-participation/client-event-participation-type';

@ObjectType({ description: 'Basic user of the system' })
@Entity()
export class Client extends User {
  @Field(() => [Event], { nullable: 'items' })
  @ManyToMany(
    type => Event,
    event => event.bookedBy,
    { lazy: true },
  )
  @JoinTable()
  public events: Lazy<Event[]>;
```

Рисунок 3.1. Частина реалізації класу Клієнт для роботи з сутністю клієнтів додатку

Крім того, необхідно відзначити, що код має дві точки входу: одну для гібридної та традиційної архітектури, та іншу для коду що розгортається у Serverless. (див. рисунок

3.6) Це зумовлено необхідністю використання різної конфігурації серверу в залежності від середовища. Окрім різних точок входу весь інший код є однаковим незалежно від середовища.

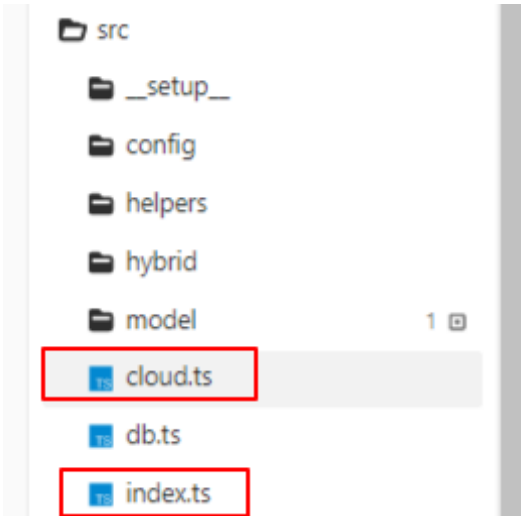


Рисунок. 3.6. Точки запуску серверу в залежності від середовища.

3.2.4. БІБЛІОТЕКИ ТА ЗАЛЕЖНОСТІ

Веб-сервер, що використовується у даному дослідженні для апробації розробленої гібридної архітектури розроблений на платформі Node.js. Node.js — платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript. Засновником платформи є Раян Дал (Ryan Dahl). Якщо раніше Javascript застосовувався для обробки даних в браузері користувача, то Node.js надав можливість виконувати JavaScript-скрипти на сервері та відправляти користувачеві результат їх виконання. Платформа Node.js перетворила JavaScript на мову загального використання з великою спільнотою розробників.

Node.js має наступні властивості:

- асинхронна одно-нитева модель виконання запитів;
- неблокуючий ввід/вивід;
- система модулів CommonJS;
- рушій JavaScript Google V8;

Для керування модулями використовується пакетний менеджер npm (node package manager).

Проект був реалізовований з використанням мови TypeScript. TypeScript — мова програмування, представлена Microsoft восени 2012; позиціонується як засіб розробки веб-застосунків, що розширює можливості JavaScript Розробником мови TypeScript є Андерс Гейлсберг (англ. Anders Hejlsberg), який створив раніше C#, Turbo Pascal і Delphi.

TypeScript є зворотньо сумісним з JavaScript. Фактично, після компіляції програму на TypeScript можна виконувати в будь-якому сучасному браузері або використовувати спільно з серверною платформою Node.js.

Переваги над JavaScript:

- можливість явного визначення типів (статична типізація),
- підтримка використання повноцінних класів (як в традиційних об'єктно-орієнтованих мовах),

- підтримка підключення модулів.

За задумом ці нововведення мають підвищити швидкість розробки, прочитність, рефакторинг і повторне використання коду, здійснювати пошук помилок на етапі розробки та компіляції, а також швидкодію програм. Основний принцип мови — весь існуючий код на JavaScript сумісний з TypeScript, тобто в програмах на TypeScript можна використовувати стандартні JavaScript-бібліотеки і раніше створені напрацювання. Більш того, можна залишити існуючі JavaScript-проекти в незмінному вигляді, а дані про типізації розмістити у вигляді анотацій, які можна помістити в окремі файли, які не заважатимуть розробці і прямому використанню проекту (наприклад, подібний підхід зручний при розробці JavaScript-бібліотек).

Для спрощення розробки у проекті було використано численні бібліотеки з відкритим вихідним кодом. Серед них варто відзначити наступні:

- TypeGraphQL (див. Розділ 3.2.3).
- TypeORM (див. Розділ 3.2.3).
- Express.js – програмний каркас розробки серверної частини веб-застосунків для Node.js, реалізований як вільне і відкрите програмне забезпечення під ліцензією MIT. Він спроектований для створення веб-застосунків і API. Де-факто є стандартним каркасом для Node.js.
- Apollo Server – це найкращий спосіб швидко створити готовий до застосування API для самостійного документування для клієнтів GraphQL, використовуючи дані з будь-якого джерела. Це бібліотека з відкритим кодом і чудово працює як окремий сервер, доповнення до існуючого HTTP-сервера Node.js або в середовищі Serverless.
- express-jwt – програмне забезпечення посередник, що валідує JsonWebTokens і встановлює користувача, що здійснює запит. Цей модуль дозволяє аутентифікувати запити HTTP, використовуючи маркери JWT у своїх програмах Node.js. JWT зазвичай використовуються для захисту кінцевих точок API та часто видається за допомогою OpenID Connect.

- typedi – це інструмент впровадження залежності для JavaScript та TypeScript. Використання TypeDI дозволяє створювати добре структуровані та легко тестовані програми.
- Ramda – практична функціональна бібліотека для програмістів JavaScript.
- Jest – фреймворк тестування JavaScript з акцентом на простоту
- Eslint – це інструмент аналізу статичного коду для виявлення проблемних зразків, знайдених у коді JavaScript. Правила в ESLint можна настроювати, а спеціальні правила можна визначати та завантажувати. ESLint охоплює як якість коду, так і проблеми стилю кодування. Код написаний за допомогою JSX або TypeScript також може бути оброблений, коли використовується плагін або транспілер.

3.2.5. GRAPHQL API

У Розділі 1.2 було детально розглянути побудову API за допомогою мови запитів GraphQL та здійснено порівняння даного підходу з використанням класичного архітектурного стилю REST. У додатку-прототипу було реалізовано API на основі мови запитів GraphQL. Основними функціональними можливостями API є (див. Додатки 1-2):

- автентифікація;
- отримання списку Клієнтів;
- отримання списку Подій;
- отримання списку Локацій:
- додавання Події;
- додавання Клієнта (реєстрація Користувача);
- бронювання Події;
- додавання запису про участь Клієнта у Події.

```
const userQuery = gql`
  query($_id: Int) {
    client(_id: $_id) {
      _id
      name
      participations {
        _id
        note
        event {
          title
          startDate
          specialist {
            name
          }
          location {
            title
          }
        }
      }
      futureEvents {
        _id
        title
        startDate
      }
    }
  }
`;
```

Рисунок. 3.7. Використання мови запитів GraphQL для вивантаження комплексної інформації одним запитом.

Крім цього важливо відзначити, що GraphQL надає можливість отримати інформацію зі складною структурою за один запит. Наприклад, для завантаження інформації про користувача, його заброньовані та потенційні події, інформацію про ці поді, спеціалістів, що їх проводять, місце проведення події можна скористатися одним запитом (див. рисунок 3.7).

Під час виконання запиту веб-сервер здійснить декілька запитів в базу даних для отримання необхідної інформації (див. рисунок 3.8).

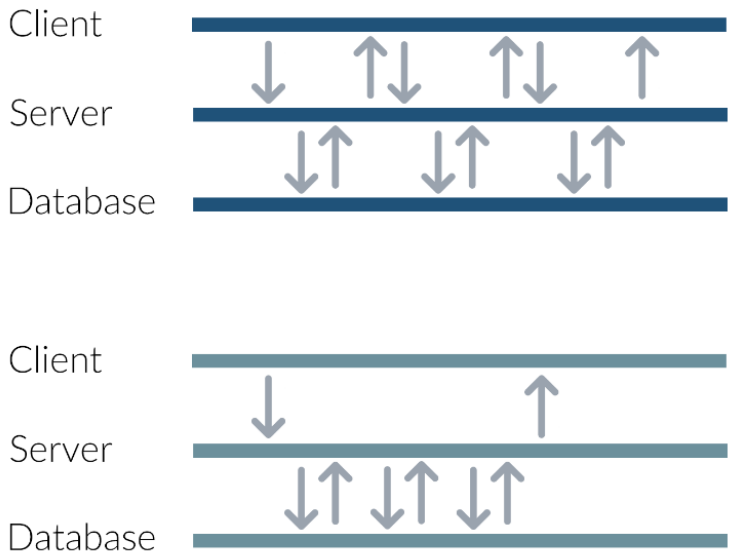


Рисунок. 3.8. Порівняння традиційної (вгорі) та GraphQL(внизу) клієнт-серверної взаємодії

Крім точок доступу, що реалізують власне бізнес-логіку додатку, було також розроблено додатковий елемент API, що здійснює симуляцію виконання задачі що потребує певних обчислювальних ресурсів. Дана точка API необхідна для перевірки продуктивності архітектури застосунку під час виконання ресурсоємних задач (див. Розділ 4.2.).

3.3. КОМПІЛЯЦІЯ ТА РОЗГОРТАННЯ ЗАСТОСУНКУ

Компіляція веб-серверу розробленого у даному досліджені передбачає перетворення коду написаного на мові TypeScript у мову JavaScript, що може бути виконана у середовищі Node.js. Для здійснення компіляції застосовується стандартний компілятор «tsc», тому необхідно виконати всього одну команду – «npm run tsc». Важливо, однак, відзначити що для компіляції необхідно створити конфіг-файл, де вказано основні параметри. (див. рисунок 3.2)

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "lib": ["es2016", "esnext.asynciterable"],
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "esModuleInterop": true,
    "noImplicitAny": true,
    "moduleResolution": "node",
    "sourceMap": true,
    "outDir": "dist",
    "baseUrl": ".",
    "paths": {
      "*": [
        "node_modules/*",
        "src/types/*"
      ]
    }
  },
  "include": [
    "src/**/*"
  ]
}
```

Рисунок 3.2. Конфігурація компіляції TypeScript у JavaScript

Розгортання додатку здійснюється трьома різними способами, для використання 3 різних видів архітектури – традиційної, Serverless та гібридної (див. таблицю 3.2).

Таблиця 3.2. Середовища розгортання застосунку-прототипу в залежності від архітектури

Архітектура\Середовище	Google Kubernetes Engine	Google Cloud Functions
Традиційна	Так	Ні
Serverless	Ні	Так
Гібридна	Так	Так

Для виконання розгортання було розроблено автоматизовані скрипти (див. рисунок 3.9). Крім того, автоматично також може бути розміщений проміжний менеджер пулу з'єднань для покращення масштабованості бази даних (див. розділ 3.1).

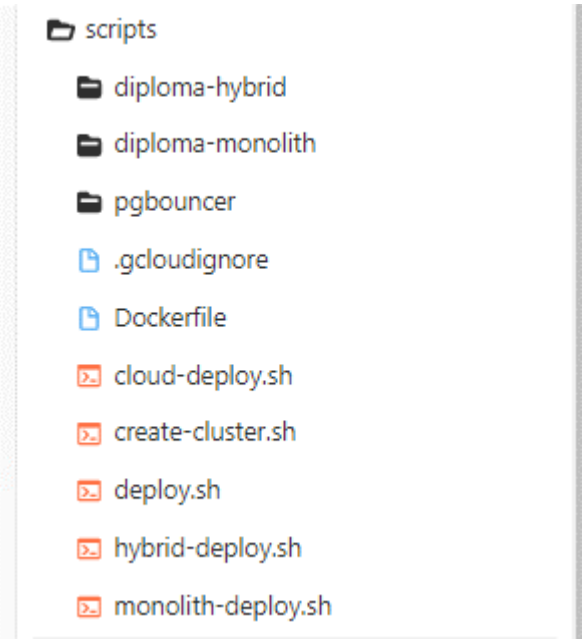


Рисунок. 3.9. Система автоматизованого розгортання додатку-прототипу

Для автоматизованого розгортання важливе значення має наявність інтерфейс командного рядка який надає провайдер послуг. Для виконання команд пов'язаних з Serverless можна використовувати «gcloud cli» - інтерфейс командного рядка gcloud - це інструмент, що забезпечує первинний CLI для Cloud Cloud Platform. Цей інструмент можна використовувати для виконання багатьох загальних завдань платформи або з командного рядка, або в сценаріях та інших автоматизаціях. Для роботи з Kubernetes

використовується Kubectl - інтерфейс командного рядка для запуску команд управління кластерами Kubernetes.

Певну складність створює необхідність одночасного розгортання гібридної архітектури у двох середовищах. Враховуючи, що гібридна архітектура передбачає існування вже розгорнутої Serverless архітектури, спочатку здійснюється розгортання Serverless, а вже потім основного серверу. У рисуноку 3.3 наведено приклад коду, що здійснює поетапне розгортання гібридної архітектури.

```
1  CLOUD_URL=$(./cloud-deploy.sh) || exit 1
2  echo "Deployed cloud functions to $CLOUD_URL"
3
4  gcloud builds submit --tag gcr.io/diploma-257315/diploma .
5
6  kubectl create configmap diploma-hybrid \
7    --from-literal DB_HOST="34.67.236.92" \
8    --from-literal REDIRECT_TO_CLOUD=true \
9    --from-literal CLOUD_URL="$CLOUD_URL" \
10   --from-literal REDIRECT_TO_CLOUD_THRESHOLD=7 \
11   --from-literal ARRAY_LENGTH=30000 \
12   -o yaml \
13   --dry-run | kubectl replace -f -
14
15 kubectl apply -f ./diploma-hybrid/deployment.yaml
16 kubectl apply -f ./diploma-hybrid/service.yaml
17 kubectl apply -f ./diploma-hybrid/autoscaler.yml
18 kubectl delete pod -l app=diploma-hybrid
```

Рисунок 3.3. Поетапне розгортання гібридної архітектури. Спочатку здійснюється розгортання Serverless (строка 1) потім результат розгортання використовується як параметр для основного серверу.

3.4. МОНІТОРИНГ ТА СУПРОВІД ДОДАТКУ

Через розподілений характер мережі Інтернет, збої можуть траплятися в багатьох точках, деякі з яких поза контролем веб-розробника. Наявність можливості швидко встановити, які компоненти функціонують, а які вийшли з ладу, може значно скоротити час, необхідний для діагностики проблеми. Після створення резервного копіювання та функціонування веб-сайту дані можуть бути використані для інформування клієнтів та управління причинами даної несправності.

Моніторинг може запобігти проблемам, виявивши схеми використання ресурсів та продуктивності, які в іншому випадку можуть залишитися непоміченими. Наприклад, повні диски зазвичай викликають безліч проблем для додатків та операційних систем. Просто знання того, що диск наближається до межі можливостей, може заощадити веб-розробнику години на виправлення проблем, викликаних повним диском.

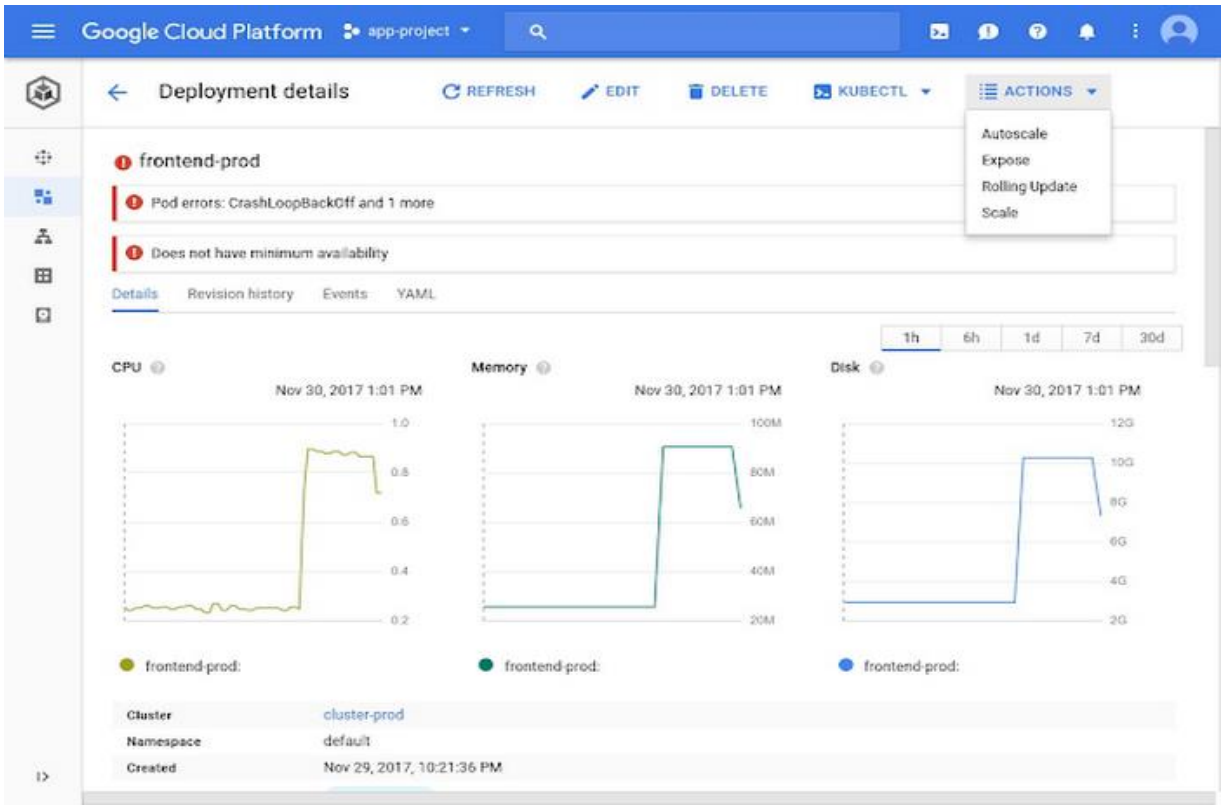


Рисунок. 3.10. Система моніторингу Google Kubernetes Engine

Провайдери хмарної інфраструктури надають засоби моніторингу стану веб-серверу. Головними показниками які можна відстежити є завантаження процесора, використання пам’яті та сховища даних (див. рисунок 2.5, 3.10).

Логування може бути здійснене як стандартними засобами STDOUT та STDERR так і з використання сторонніх платформ та бібліотек. Провайдер Google надає зручний інструмент для роботи з логами - Google Stackdriver. Це безкоштовна послуга управління системами хмарних обчислень, пропонована Google. Він надає дані про продуктивність та діагностику (у вигляді моніторингу, ведення журналів, відстеження, повідомлень про помилки та оповіщення) для публічних користувачів хмари. Stackdriver - це гібридне хмарне рішення, що забезпечує підтримку як хмарних середовищ Google, так і AWS (див. рисунок 3.11).

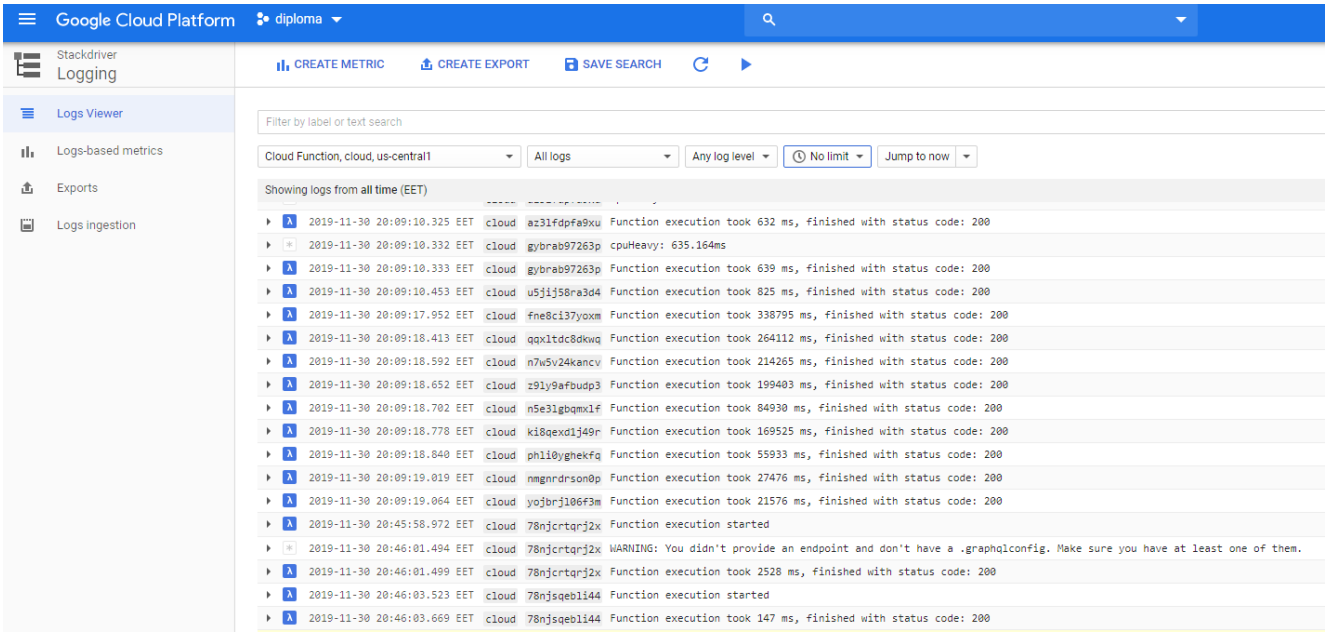


Рисунок. 3.11. Перегляд логів у системі Stackdriver

ВИСНОВКИ ДО РОЗДІЛУ 3

У результаті аналізу переваг та недоліків традиційної та Serverless архітектури було здійснено наступне:

1. Запропоновано поєднати традиційну та Serverless архітектуру з метою нівелювання недоліків та збереження переваг обох підходів.

2. Гібридна архітектура заснована на традиційній, однак, за необхідності, передбачає можливість автоматичного залучення додаткових обчислювальних ресурсів шляхом звернення до інфраструктури Serverless.

3. Для аналізу характеристик розробленої архітектури було створено додаток-прототип. Застосунок реалізує систему обслуговування процесу надання послуг клієнтам для певного бізнесу і надає можливість забронювати отримання певної послуги у певного спеціаліста у певній локації.

4. Для роботи з базою даних, гібридна архітектура передбачає використання менеджеру пулу з'єднань – серверу, що має низькі вимоги до продуктивності апаратного забезпечення і єдиним завданням якого є розпоряджання пулом з'єднань до сервісу постійного зберігання даних (бази даних). Необхідний для забезпечення стійкого зв'язку між обчислювальними потужностями та базою даних, враховуючи обмежену кількість підключень до бази даних, що може бути встановлена одночасно.

5. У додатку-прототипу було реалізовано API на основі мови запитів GraphQL.

6. Розгортання додатку здійснюється трьома різними способами, для використання трьох різних видів архітектури – традиційної, Serverless та гібридної.

7. Моніторинг та супровід додатку здійснюється Google Stackdriver. Це безкоштовна послуга управління системами хмарних обчислень, пропонована Google.

4. ТЕСТУВАННЯ ПРОДУКТИВНОСТІ ТА ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ ПРОТОТИПУ ДОДАТКУ НА SERVERLESS ТА GRAPHQL

4.1. ВИЗНАЧЕННЯ ПРОДУКТИВНОСТІ ТА ЕФЕКТИВНОСТІ ВЕБ СЕРВЕРУ

Хоча людям властиво помилятися, іноді вартість помилки може бути занадто високою. Історія знає безліч прикладів ситуацій, коли недоліки програмного забезпечення можуть спричинити збитки у розмірі мільярдів доларів або навіть призвести до жертв: починаючи кав’ярнями Starbucks, що були змушені роздавати безкоштовні напої через несправність у реєстрі, закінчуючи військовим літаком F-35, що не може виявити цілі правильно через несправність радіолокатора.

Щоб переконатися, що випущене програмне забезпечення є безпечним та функціонує, як очікувалося, було введено поняття якості програмного забезпечення. Його часто визначають як "ступінь відповідності явним або неявним вимогам та очікуванням" [29]

Ці так звані явні та неявні очікування відповідають двом базовим рівням якості програмного забезпечення:

- Функціональний - відповідність продукту функціональним (явним) вимогам та технічним умовам. Цей аспект орієнтований на практичне використання програмного забезпечення з точки зору користувача: його особливості, продуктивність, зручність у використанні, відсутність дефектів.
- Нефункціональні - внутрішні характеристики та архітектура системи, тобто структурні (неявні) вимоги. Сюди входить ремонтопридатність, зрозумілість, ефективність, продуктивність та безпека. [30]

Оцінка якості ПЗ – сукупність операцій, які включають вибір номенклатури показників якості оцінюваного ПЗ, визначення значень цих показників і порівняння їх з базовими значеннями. [29]

Таблиця 4.1. Метрики продуктивності

Параметр	**Одиниці виміру**	**Опис**
Час відповіді	Час (мілісекунди)	Час очікування, поки клієнт отримає відповідь сервера після надсилання запиту.
Пропускна здатність	запит/секунда	Кількість запитів, які обробляються додатком за одиницю часу
Використання ресурсів	відсоток	Використання ресурсів, таких як ЦП (центральний процесор), пам'ять, пропускна здатність мережі

У даному дослідження запропонована гібридна архітектура буде порівняна з традиційною хмарною та архітектурою на основі виключно Serverless за допомогою критерії нефункціонального характеру – продуктивності та ефективності.

У даному досліджені було викормлено перелік характеристик, що можна віднести до ефективності програмного забезпечення, в тому числі, веб-серверу (див. таблицю 4.2).

Таблиця 4.2. Метрики ефективності

Параметр	**Одиниці виміру**	**Опис**
Кількість помилок	відсоток	Кількість помилок на 100 запитів
Вартість запиту	у. о. / 1 000 000 запитів	Грошове вираження вартості користування системою
Затрати на обслуговування	людино-години	Затрати на розгортання та підтримання інфраструктури

Продуктивність стосується того, наскільки швидкою є "відповідь програмного забезпечення, коли відбувається подія [31]. Події надходять до програмної системи в

різних моделях, які можна «охарактеризувати як періодичні або стохастичні» [31]. Користуючись даним визначенням було розроблено список критеріїв, які було виміряно та оцінено для визначення придатності запропонованої архітектури до використання при розробці веб-серверу (див. таблицю 4.1).

Ефективність – характеризує ступінь задоволення потреб користувача в обробці даних з урахуванням економічних, людських ресурсів і ресурсів системи обробки інформації. [32]

4.2. КОНФІГУРАЦІЯ ТЕСТІВ ТА ОГЛЯД СЦЕНАРІЇВ

Продуктивність програми-прототипу вивчається для розуміння можливості застосування безсерверної та гібридної архітектури для виробничих веб-сервісів. Тестування навантаження використовується для того, щоб переглянути роботу програми в різних сценаріях та конфігураціях. Основна мета - вивчити поведінку програми в різних умовах навантаження. Для цього використовується хмарне середовище виконання програмного забезпечення для виконання тестів. Розглянемо детальніше налаштування тестів.

Для порівняння було розроблено 3 варіації додатку прототипу. Кожен з зазначених додатків використовує менеджер пулу з’єднань «pgbouncer» для того щоб виключити дану складову з порівняння підходів:

1. Звичайна архітектура - автомасштабований Kubernetes сервер 1-3 вузла.
2. Serverless інфраструктура на основі Google Cloud Functions.
3. Гібридна - автомасштабований Kubernetes сервер 1-3 вузла та Serverless. Поєднання звичайної та безсерверної інфраструктури.

У цьому дослідженні було використано наступні спільні налаштування виконання тестів:

- 50 користувачів
- До 500 запитів в секунду
- Тривалість 10 хвилин
- 3 етапи зростання навантаження
- Пікове навантаження через 1 хвилини початку тестування (див. рисунок 3.1)
- Локація – регіон розміщення веб-серверу

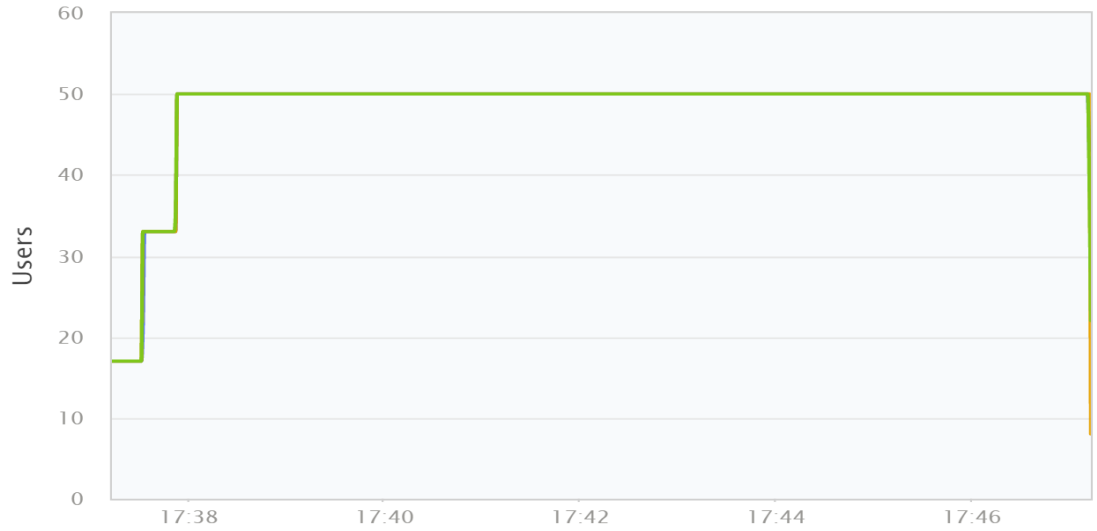


Рисунок. 3.1. Зростання кількості користувачів під час тестування

Для порівняння архітектур було виокремлено наступні сценарії:

1. **Сценарій А**. Активна робота з базою даних:

- 3 запита на читання / 1 запит на створення;
- 2 складних запита на читання / 1 простий запит на читання.

2. **Сценарій Б**. Активна робота з базою даних та запит, що вимагає обчислювальних операцій. Даний запит було розроблено з розрахунку середнього виконання близьку 100 мілісекунд на процесорі «Intel Core 7» 8-го покоління.

- 3 запита на читання / 1 запит на створення / 1 запит, що вимагає близько 100 мілісекунд обчислювального часу;
- 2 складних запита на читання / 1 простий запит на читання.

Слід зазначити обмеження щодо розподілу ресурсів для тестування навантаження. У зв’язку з жорстким бюджетом цього дослідження, автор повинен вибрати дуже базовий варіант тестування навантаження, який не дозволяє мати більше навантажень на додаток. Бюджетна підписка дозволяє залучити до тесту максимум 50 одночасних користувачів.

Наступна таблиця дає коротке пояснення для різних статистичних даних, які будуть згадані в аналізі та обговоренні результатів тестів.

Таблиця 4.3. Пояснення показників, що використовуються у результатах тестування додатків. Складено автором на основі [34]

Загальна кількість запитів	Загальна кількість запитів, надісланих до програми. Кожен запит очікує, що з програми буде повернуто одну відповідь.
Запит	Відповідно, запит - це один запит до програми.
Запит в секунду	Кількість запитів, розпочатих щосекунди.
Середня пропускна здатність (запит / секунда)	Це середня швидкість передачі даних, що надходить до програми, яка вимірюється зверненнями в секунду. Наприклад, середня пропускна здатність 50 запитів / секунда означає, що щосекунди відбувається 50 запитів.
Середній час відповіді (мілісекунд)	Час відповіді - це кількість часу, необхідного для обробки запиту, вимірюється часом, що минув між першим байтом даних відправлених користувачем та останнім байтом даних, отриманих цим користувачем. Середній час відповіді - це середнє значення таких разів для всіх запитів. Іншими словами ця статистика розкриває, як довго користувач в середньому отримує відповідь.
90% часу відгуку (мілісекунд)	Це 90-й відсоток часу відгуку, який спостерігає більшість користувачів.
Помилки (%)	Відсоток помилок у всіх запитах.

4.3. СЦЕНАРІЙ А

Виконання тестування навантаження для додатку-прототипу на основі різного середовища розгортання веб серверу було отримано результати наведені у таблиці 4.4. Результати експерименту показують, що традиційна архітектура демонструє найкращі показники продуктивності. Serverless архітектура опрацьовує на 12 % менше запитів, а гібридна на 21 %. При цьому, однак, кількість помилок отриманих користувачем є найбільшою саме при застосування традиційної архітектури.

Таблиця 4.4. Результати виконання тестового сценарію А

Архітектура	**# Кількість запитів**	**Середній час відгуку**	**90%**	**99%**	**% помилок**	**Запитів/с**
Традиційна	256 432	62.90	150	505	0.069	428.10
Serverless	230 347	97.81	190	763	0	383.91
Гібридна	204 209	141.36	197	531	0.013	340.35

На рисунку 3.1 проілюстровано, що розроблена гібридна архітектура має кращу пропускну здатність на початку тестування. Тобто завдяки використанню традиційного серверу, гібридна архітектура швидше адаптується до початкового рівня навантаження. Так само, на заваршальній стадії, гібридна архітектура показує кращі показники стабільності. Більша пропускна здатність на початку тестування може означати, що гібридна архітектура швичше адаптується до невеликих скачків у навантаженні (див. рисунок 3.1).

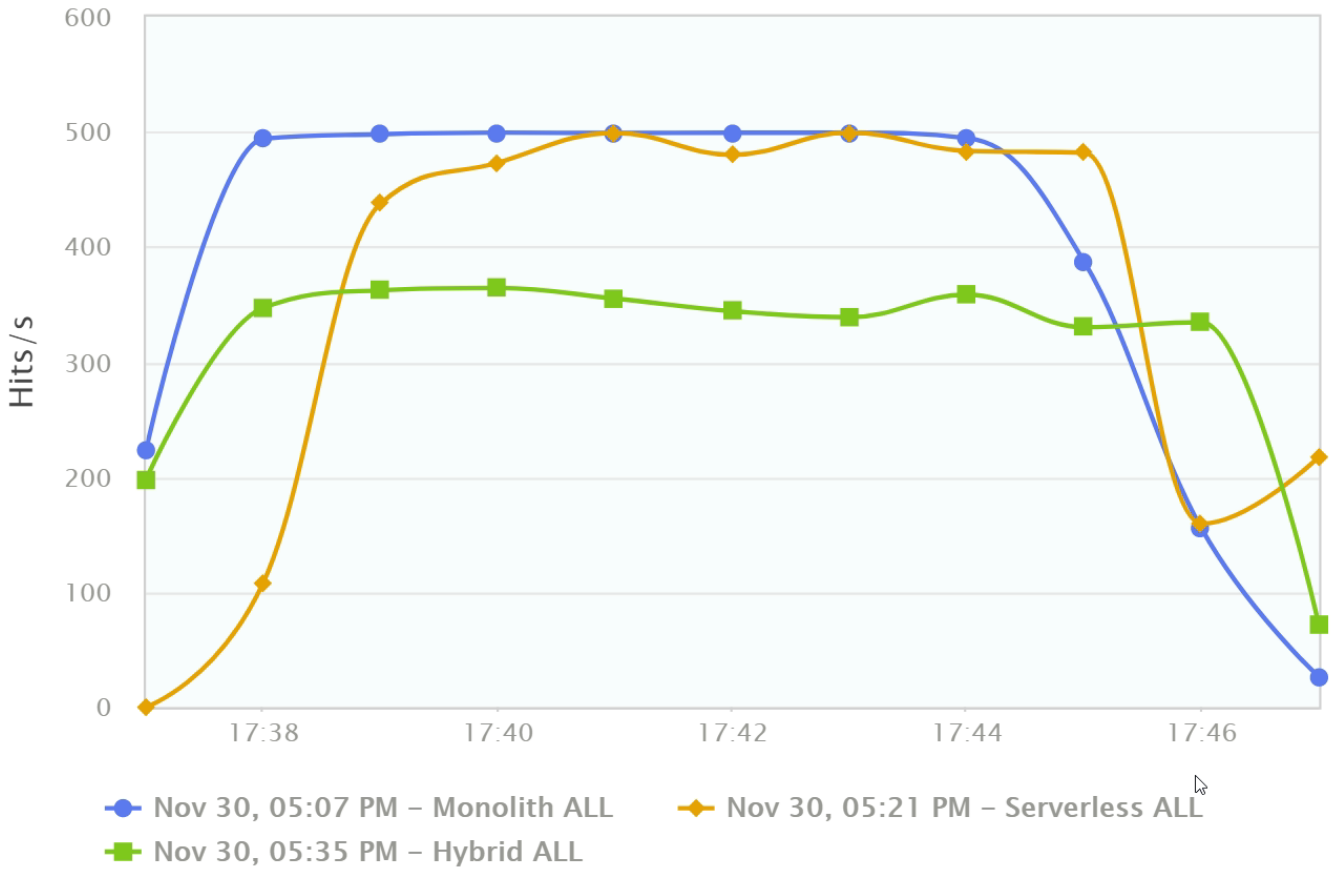


Рисунок. 3.2. Порівняння пропускної здатності традиційної, Serverless та гібридної інфраструктури для тестового Сценарію А.

Провадений аналіз засвідчує, що у рамках розрахункового навантаження доступного для проведення даного дослідження, традиційна архітектура за умови використання інфраструктури Kuberenetes з можливістю автомаштабування краще показує себе за умови виконання базових CRUD запитів з використанням мови запитів GraphQL.

4.4. СЦЕНАРІЙ Б

Розглянемо більш детально якісні показники додатку-прототипу за умови використання різних стилів архітектури за умови, що один із запитів вимагатиме значного використання обчислювальних ресурсів. У описаній ситуації, гібридна архітектура показує найкращі результати – веб-сервер на основі гібридної архітектури здатний обробити на 14 відсотків більше запитів ніж Serverless та на 55 відсотків більше запитів ніж традиційний. При цьому, традиційний сервер має значно вищий рівень помилок – 0.2 % проти 0 та 0.004 у Serverless та гібридній архітектурі відповідно.

Таблиця 4.4. Результати виконання тестового сценарію Б

Запит	**# Кількість запитів**	**Середній час відгуку**	**90%**	**99%**	**% помилок**	**Запитів/с**
Традиційна	60 180	481.09	1287	3279	0.214	100.30
Serverless	111 201	261.37	671	1783	0	185.34
Гібридна	129 885	223.52	631	1039	0.004	216.47

Розглянемо більш детально стан інфраструктури під час проведення тестування за сценарієм Б та окремі показники продуктивності отримані під час експерименту. На рисунку 3.3 показано кількість операцій виконаних базою даних під час виконання тестів (зліва на право – Serverless, гібридна, традиційна архітектура). Отримані дані показують, що гібридна архітектура більш стабільна та швидше масштабується коли сервер має окрім звернень до бази виконувати, також, і складні обчислення.

Рисунок. 3.3. Навантаження на базу даних PostgreSQL під час виконання тестів (зліва на право – Serverless, гібридна, традиційна архітектура)

На рисунку 3.4 показано кількість контейнерів Serverless, що були залучені інфраструктурою під час здійснення експерименту. Гібридною архітектурою було залучено більше обчислювальних одиниць, однак здійснено це було більш плавно, що пояснюється наявністю звичайного серверу, що теж здійснював обробку запитів.

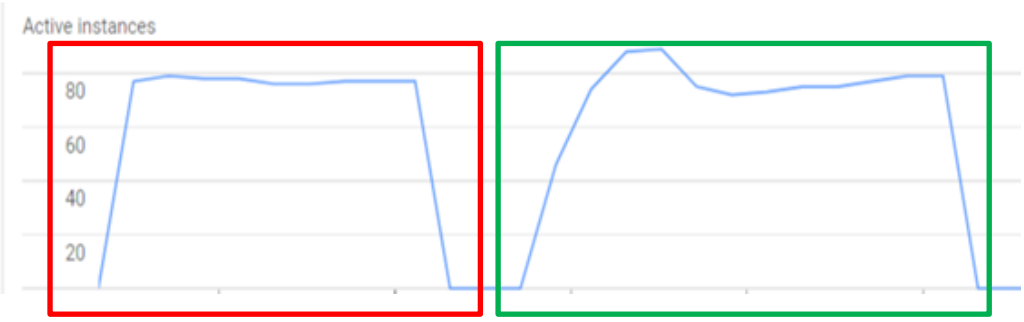


Рисунок. 3.4. Кількість контейнерів Serverless при здійсненні експерименту (зліва на право – Serverless та гібридна архітектура)

Як було зазначено у розділі 2.3, Serverless інфраструктура може достатньо повільно реагувати на зростання навантаження, що може сповільнювати час відповіді на запит до веб-серверу. На рисунку 3.5. добре помітно, що традиційна та гібридна архітектура показують значно менший час відповіді ніж Serverless.

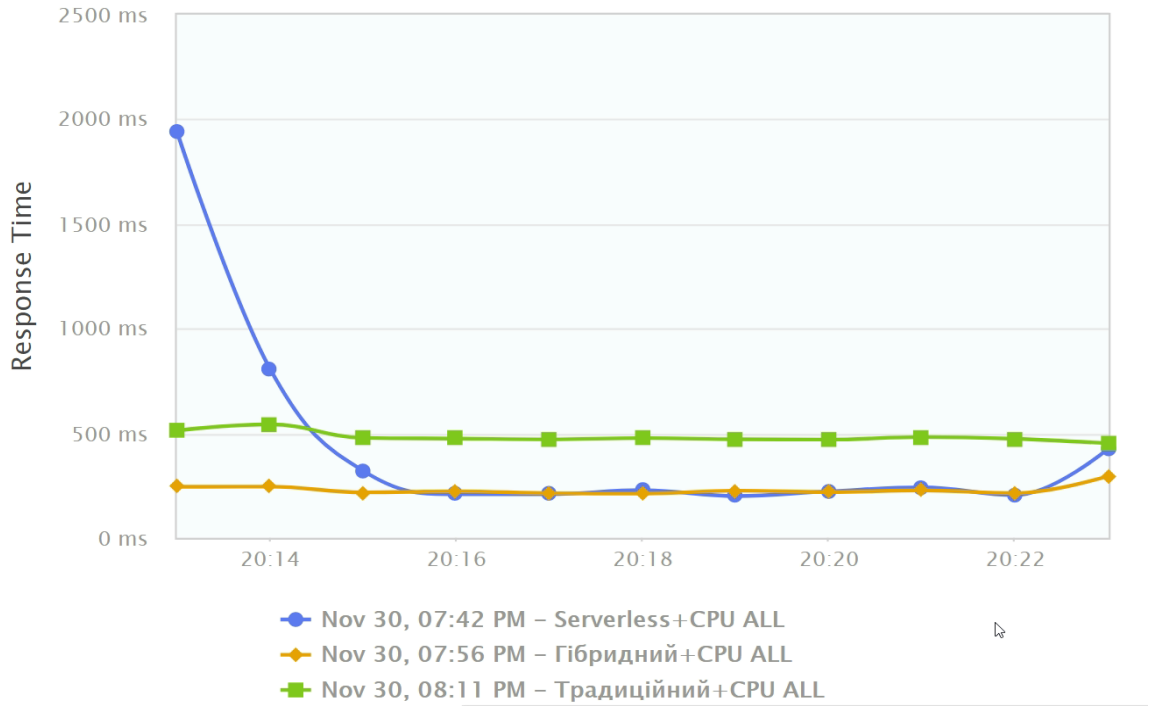


Рисунок. 3.5. Динаміка середньої тривалості відповіді на запит до веб-серверу

На рисунку 3.6 проілюстровано, що розроблена гібридна архітектура має кращу пропускну здатність. Особливо на початку тестування завдяки використанню традиційного серверу, гібридна архітектура швидше адаптується до початкового рівня навантаження.

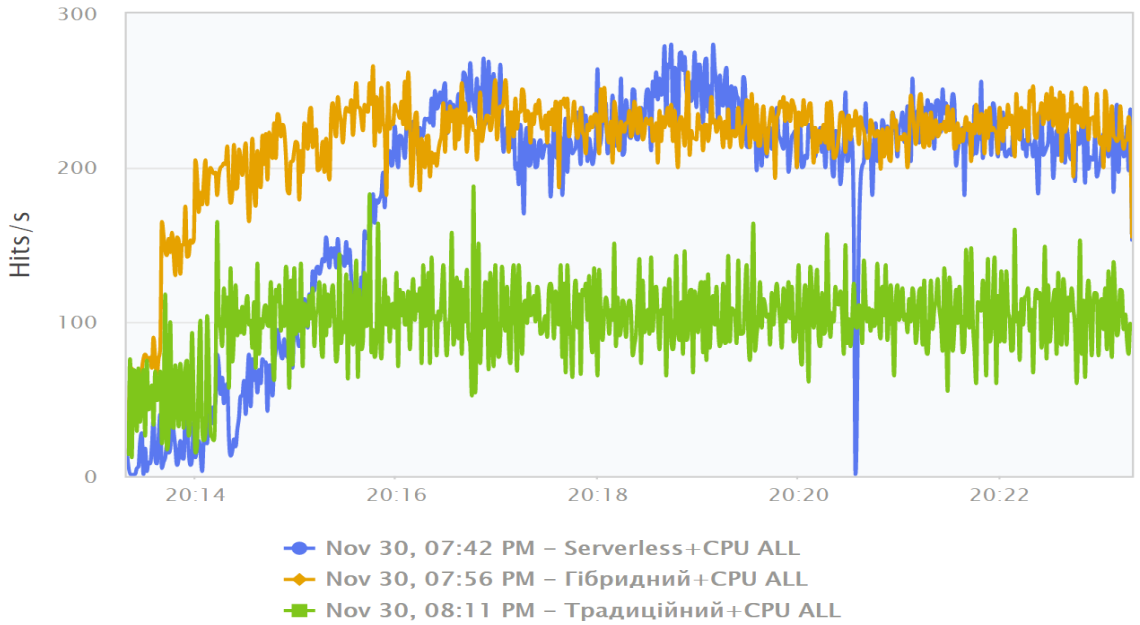


Рисунок. 3.6. Порівняння пропускної здатності традиційної, Serverless та гібридної інфраструктури для тестового Сценарію Б

4.5. ОЦІНКА ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНИХ ПІДХОДІВ

Для розрахунку вартості запиту було взято до уваги затрати на утримання наступних складових інфраструктури:

- база даних Cloud SQL PostgreSQL;
- фвтоматичне балансування навантаження Google Load Balancer;
- Kubernetes сервер для менеджера пулу з’єднань pgbouncer;
- Kubernetes сервер
- Serverless Google Cloud Functions;

Крім того було здійснено декілька припущень:

- рівномірне навантаження серверу протягом періоду часу, що береться до розрахунку;
- середня кількість запитів – 100 запитів у секунду;
- близько 20 % запитів буде оброблено традиційним сервером за умови використання гібридної архітектури.

Таблиця 4.5. Результати оцінки ефективності запропонованих підходів до розробки архітектури веб-серверу

Параметр\Архітектура	**Сценарій А**			**Сценарій Б**		
	Звичайна	**Serverless**	**Гібридна**	**Звичайна**	**Serverless**	**Гібридна**
Кількість помилок	0.069%	0	0.013%	0,214 %	0%	0,004%
Вартість запиту	$0.45	$0.5	$0.6	$0.45	$0.5	$0.6
Затрати на обслуговування	Більші	Менші	Найбільші	Більші	Менші	Найбільші

У результаті здійсненого аналізу виявлено, що традиційна архітектура має найнижчу пряму вартість запиту, однак і найбільшу кількість помилок (див. таблицю 4.5).

Затрати на обслуговування було розраховано шляхом припущень, що Serverless допомагає вирішити ряд проблем, що зазвичай вимагає тривалого часу та значних зусиль (див. розділ 2.3). Відповідно, Serverless має найменше значення затрат людино-годин розробників веб-серверу, а гібридна архітектура найбільше, оскільки вимагає підтримки двох видів інфраструктури.

4.6. РЕКОМЕНДАЦІЇ ЩОДО ЗАСТОСУВАННЯ SERVERLESS ДЛЯ ПОБУДОВИ ВЕБ-СЕРВЕРУ

Оскільки будівельними блоками Serveress програми є прості функції, немає необхідності боротися з усіма інфраструктурними проблемами - завдання розробника просто писати та розгортати програму. Звичайно, розробнику все одно доведеться розібратися в тому, як налагодити спілкування між функціями та клієнтами, зберігати стан і працювати з базою даних.

Враховуючи результати проведеної оцінки показників продуктивності та ефективності веб-серверу побудованого як на Serverless, так і з використанням гібридної архітектури, необхідно зазначити, що Serverless додатки є повільнішими за умови середнього стабільного навантаження, коли мова йде про застосунки, що в основному виконують операції вводу виводу даних з певного сховища (бази даних). При цьому використання стилю Serverless значно підвищує продуктивність додатків, що повинні також здійснювати певні обчислення. Останнє є наслідком того, що Serverless має необмежений обчислювальний потенціал.

Враховуючи переваги та недоліки підходу Serverless, є кілька типів програм, які найкраще підходять для даної архітектури:

- фонові завдання з високою затримкою, такі як мультимедіа або обробка даних;
- клієнт-важкі програми, де більша частина логіки може бути переміщена до клієнта;
- програми з непередбачуваним обсягом завантаження ресурсів сервера;
- швидкозростаючі та мінливі додатки, які повинні масштабуватися відразу і мати можливість швидко змінювати власні характеристики;
- веб-сервери, що потребують значних обчислювальних ресурсів для виконання запитів від користувачів.

ВИСНОВКИ ДО РОЗДІЛУ 4

Таким чином, тестування продуктивності та ефективності використання прототипу додатку на Serverless та GraphQL проведене шляхом порівняння запропонованої гібридної архітектури з традиційною хмарною та архітектурою на основі виключно Serverless дозволило досягти наступних результатів:

1. Дано визначення поняттю продуктивності та ефективності. Було виокремлено конкретні критерії вимірювання продуктивності та ефективності веб-серверу.

2. Для порівняння було розроблено 3 варіації додатку прототипу: звичайна архітектура, Serverless інфраструктура та гібридна - автомасштабований Kubernetes сервер 1-3 вузла та Serverless.

3. Для порівняння архітектур було виокремлено наступні сценарії: **Сценарій А**. Активна робота з базою даних; **Сценарій Б**. Активна робота з базою даних та запит, що вимагає обчислювальних операцій.

4. Для Сценарію А результати експерименту показують, що традиційна архітектура демонструє найкращі показники продуктивності. Serverless архітектура опрацьовує на 12 % менше запитів, а гібридна на 21 %. При цьому, однак, кількість помилок отриманих користувачем є найбільшою саме при застосування традиційної архітектури.

5. Для Сценарію Б гібридна архітектура показує найкращі результати – веб-сервер на основі гібридної архітектури здатний обробити на 14 відсотків більше запитів ніж Serverless та на 55 відсотків більше запитів ніж традиційний. При цьому, традиційний сервер має значно вищий рівень помилок – 0.2 % проти 0 та 0.004 у Serverless та гібридній архітектурі відповідно.

6. Виявлено, що традиційна архітектура має найнижчу пряму вартість запиту, однак і найбільшу кількість помилок за умови рівномірного навантаження.

7. Враховуючи результати проведеної оцінки показників продуктивності та ефективності веб-серверу побудованого як на Serverless, так і з використанням гібридної архітектури, необхідно зазначити, що Serverless додатки є повільнішими за умови середнього стабільного навантаження, коли мова йде про застосунки, що в основному виконують операції вводу виводу даних з певного сховища (бази даних). При цьому використання стилю Serverless значно підвищує продуктивність додатків, що повинні також здійснювати певні обчислення.

8. Виокремлено види програм веб-серверу, що найкраще підходять для застосування Serverless або гібридної інфраструктури

5. СТАРТАП-ПРОЕКТ

У рамках даного дослідження було розроблено проект стартапу, що спрямований на створення платформи, що полегшує кооперацію людей для участі у групових заходах (див. таблицю 5.1). Даний проект може бути розроблений шляхом розширення та доопрацювання функціоналу додатку-прототипу описаному у розділі 3.2. Застосування гібридної архітектури з використанням технології Serverless забезпечить технічну основу надійності та масштабованості даного застосунку. Крім того, використання Serverless зменшує витрати на ранніх етапах існування проекту, що також є економічною перевагою.

Таблиця 5.1. Опис ідеї стартап-проекту

Зміст ідеї	**Напрямки застосування**	**Вигоди для користувача**
Платформа, що полегшує кооперацію людей для участі у групових заходах	Об’єднати людей для спільного дозвілля: • футбол • теніс • волейбол • йога • лото • літературні вечори • тощо	1. У зручному місці в зручний час можна записатися та прийняти участь у бажаному дозвіллі. 2. Система постановки особистих цілей та відслідковування прогресу у їх досягненні. 3. Надавачі послуг зможуть надійно проінформувати користувачів про свою компанію.

Стартап спрямований на вирішення численних проблем, що мають місце у забезпеченні дозвілля у великих містах:

- слабкість інформаційних систем – системи організації застарілі або відсутні,

- тренд до саморозвитку – спорт та інші групові заняття сприяють розвитку особистості,
- складно просувати послуги – через інформаційний шум тяжко просувати певні послуги в неспеціалізованому середовищі;
- тренд до здорового способу життя – сучасні мешканці міст намагаються регулярно займатися спортом;
- урбанізація – людям тяжко об’єднуватися в групи через густонаселеність та відірваність від рідних місць.

Таблиця 5.2. Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	**Техніко-економічні характеристики ідеї**	**(потенційні) товари/концепції конкурентів**			**W**	**N**	**S**
		Мій проект	Rooky	DIKIDI			
1.	Сфери дозвілля	Всі сфери дозвілля, з фокусом на спорт	Орієнтація на сферу краси	Орієнтація на сферу краси			Так
2.	Охоплення	Найбільші міста	Локальне	Локальне			Так
3.	Інтерфейс	Зручний сучасний інтерфейс	Застарілий інтерфейс	Застарілий інтерфейс			Так

Бізнес модель продукту передбачає наступні способи отримання доходу, що наведені у таблиці 5.3. Головним джерелом доходу слугуватиме підписка отримана від користувачів.

Таблиця 5.3. Головні джерела доходів стартап-проекту

Джерело доходу	**Опис**
Підписка	Надавачі послуг повинні оплачувати користування системою на основі терміну або за кількістю відвідувань
Таргетована реклама	Надавачі послуг можуть здійснювати рекламу в системі
Глобальні заходи	Fields проводять регулярні тематичні заходи серед учасників системи

Програмний продукт стартап проекту передбачає додатки для користувачів та компаній. Застосунки повинні включати мобільний додаток, веб-портал, адмін-панель. На даний момент всі можливості для технологічної реалізації проекту існують.

Перевагами продукту перед конкурентами слугуватиме наступне:

- user-first підхід – задоволення потреб користувачів для закріплення потреби у системі;
- команда – професійні маркетологи, фінансисти, дизайнери та розробники;
- дослідження та аналіз – UI та UX побудований на основі провідних досліджень в галузі.

Маркетинг та продажі. У рамках даного стартап проекту передбачають два головні напрямки розповсюдження додатку – користувачі та надавачі послу. Нижче, на рисунку 5.1 наведено опис маркетингових заходів стартап проекту.

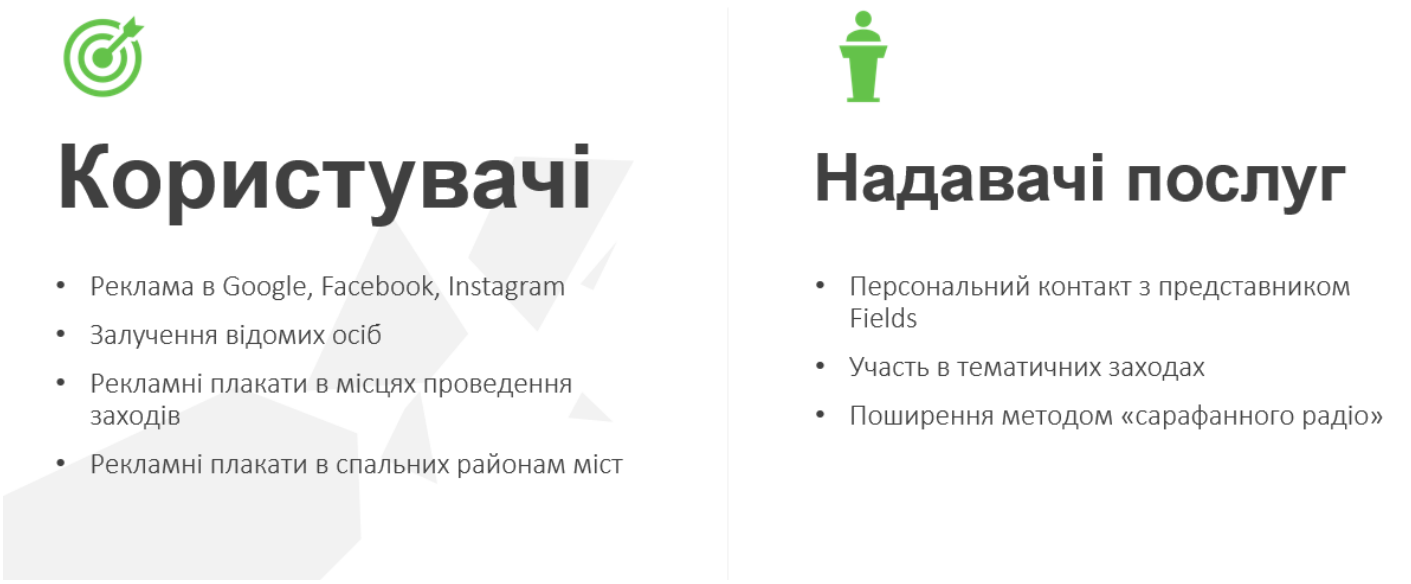


Рисунок. 5.1. Маркетинг та продажі стартап проекту

Визначення ринкових можливостей, які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту, дозволяє спланувати напрями розвитку проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів-конкурентів. У таблиці 5.4 наведені результати потенційного ринку стартап-проекту. За результатами аналізу можна стверджувати про те, що проект є привабливим.

Таблиця 5.3. Головні джерела доходів стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	3
2	Загальний обсяг продаж, грн	10 000 000
3	Динаміка ринку (якісна оцінка)	зростає
4	Наявність обмежень для входу (вказати характер обмежень)	відсутні

5	Специфічні вимоги до стандартизації та сертифікації	відсутні
6	Середня норма рентабельності в галузі (або по ринку), %	10 %

Для впровадження проекту пропонується агресивна стратегія розвитку спрямована на активну експансію спочатку на найбільші міста країни, а потім і закордон. На рисунку 5.2 наведені основні етапи розвитку проекту на найближчі 3 роки. Найголовнішими досягненнями має бути розробка MVP, запуск пілотних проектів, досягнення відмітки у 1000 активних користувачів, охоплення більшості локацій та міст, IPO на міжнародному фондовому ринку.

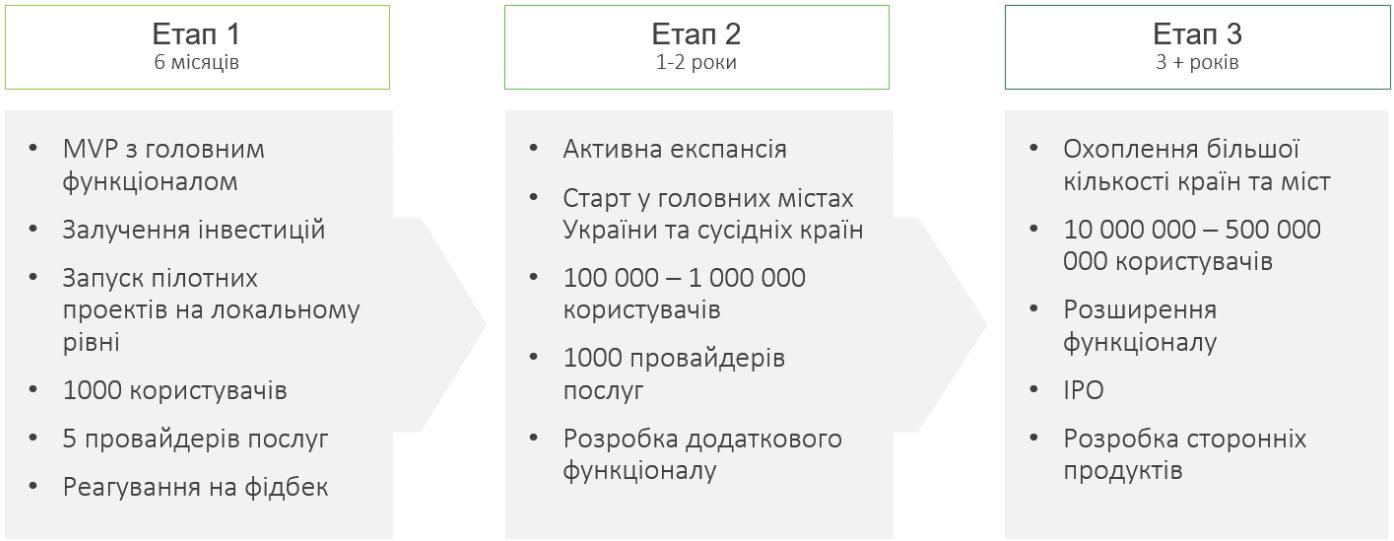


Рисунок. 5.2. Стратегія розвитку стартап-проекту

При цьому протягом найближчих 2 років планується розробка пілотного проекту, збір відгуків користувачів, старт рекламної кампанії та вихід на широкий ринок (див. рисунок 5.3).

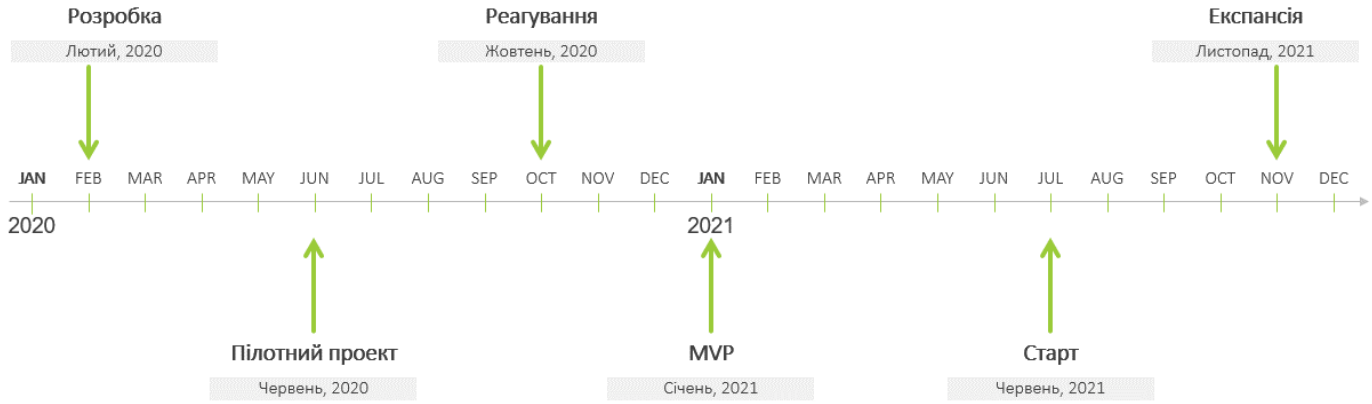


Рисунок. 5.3. Хронологія стартап-проекту на найближчі 2 роки

Для будь-якого стартап-проекту важливе значення має фінансування. Для реалізації першого етапу, даний проект потребує близько 140 тисяч доларів США. Покриття даних ресурсів планується шляхом залучення власних коштів засновників, кредитів підрядників, коштів інвесторів, а також банківського кредитування та лізингу (див. рисунок).

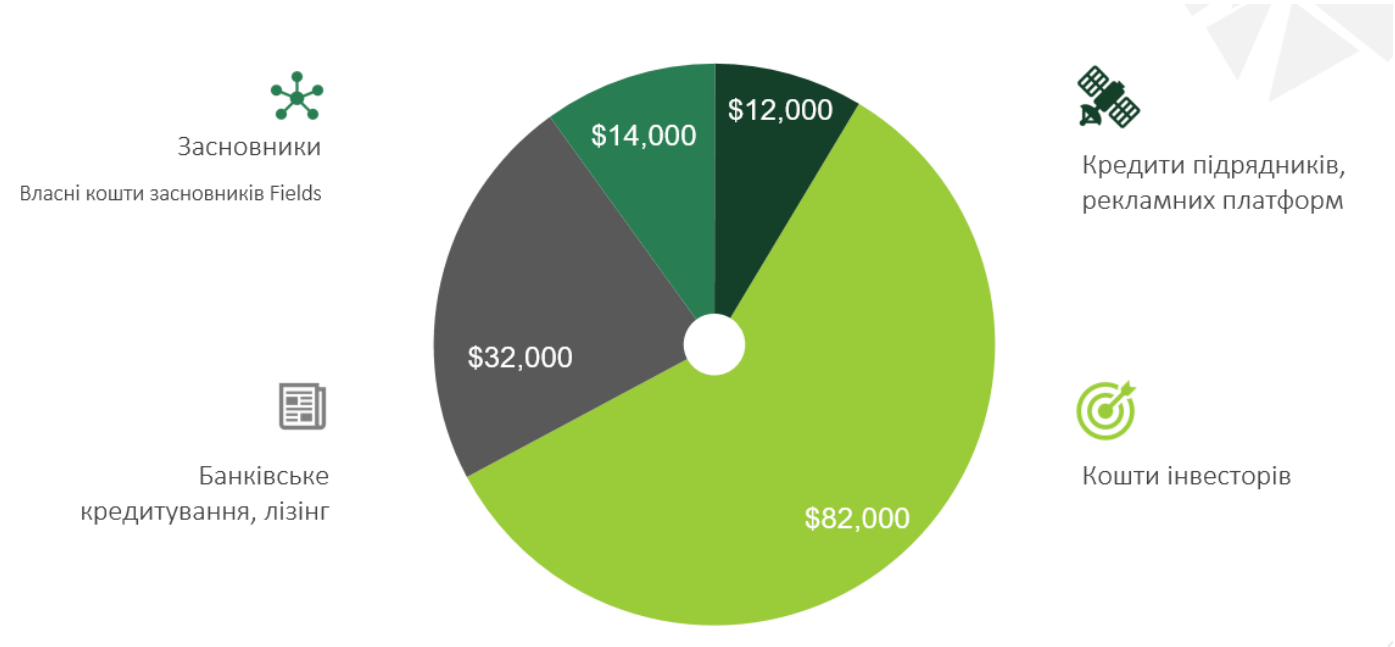


Рисунок. 5.4. Фінансування першого етапу стартап-проекту

ВИСНОВКИ ДО РОЗДІЛУ 5

У рамках даного дослідження було розроблено проект стартапу, що спрямований на створення платформи, що полегшує кооперацію людей для участі у групових заходах. Даний проект може бути розроблений шляхом розширення та доопрацювання функціоналу додатку-прототипу описаному у розділі 3.2. Застосування гібридної архітектури з використанням технології Serverless забезпечить технічну основу надійності та масштабованості даного застосунку. Крім того, використання Serverless зменшує витрати на ранніх етапах існування проекту, що також є економічною перевагою.

У результаті проведеного аналізу було розроблено основні засади стартап-проекту, щодо побудови платформи, що полегшує кооперацію людей для участі у групових заходах. Було здійснено опис ідеї проекту, технологічний аудит ідеї проекту, аналіз ринкових можливостей запуску стартап-проекту, розроблено ринкову стратегію проекту та маркетингову програму стартап-проекту.

Проведений аналіз показав, що:

- є можливість ринкової комерціалізації проекту (наявний попит, динаміка ринку, рентабельність роботи на ринку);
- є перспективи впровадження з огляду на потенційні групи клієнтів, бар’єри входження, стан конкуренції, конкурентоспроможність проекту;
- є доцільною подальша імплементація проекту.

ВИСНОВКИ

1. Досліджено поняття архітектури веб-серверу та еволюцію підходів побудови середовищ розгортання веб-серверу. Технологія Serverless є одним з найновіших засобів розгортання веб-серверу на хмарній інфраструктурі.

2. Проведено аналіз випадків можливості та доцільності застосування Serverless (безсерверних обчислень) для побудови веб-серверу. Безсерверні обчислення спрямовані на автоматизацію вирішення таких складних завдань як масштабування обчислювальних ресурсів відповідно до поточних вимог, відмова від виділення надлишкових обчислювальних ресурсів для побудови веб-серверу та інших.

Serverless використовує той факт, що звичайні сервери переважно використовують лише 5-15% своєї максимальної обчислювальної потужності. Немає необхідності мати сервер, що працює 24/7 нон-стоп, коли достатньо тільки запустити деякий програмний код, саме коли це необхідно. Це дозволяє постачальникам FaaS стягувати плату лише за той час, коли функція фактично працювала. Таким чином Serverless дозволяє використовувати інфраструктуру більш ефективно.

3. Проведено порівняльний аналіз традиційної та Serverless архітектури веб-серверу, виявити переваги та недоліки застосування безсерверних обчислень у веб-середовищі. головними відмінностями між без серверними і традиційними обчисленнями є:

- слабко зв'язані обчислення та зберігання. Масштабування зберігання та обчислення є окремим - резервується та оцінюється незалежно. Загалом, зберігання забезпечується окремою хмарною службою, а обчислення не має збереженого стану.
- виконання коду без управління ресурсами. Замість того, щоб запитувати ресурси, користувач надає фрагмент коду, а хмара автоматично надає ресурси для виконання цього коду.

- оплата здійснюється пропорційно використовуваним ресурсам замість ресурсів, що резервуються. Платежі залежать від певних параметрів виконання програми, наприклад, часом виконання. На відміну від потужності та кількості виділених віртуальних машин.

4. Для усунення недоліків Serverless інфраструктури за рахунок традиційної було розроблено гібридну архітектуру веб-серверу. Гібридна архітектура – це поєднання традиційної хмарної інфраструктури з Serverless з метою вирішення таких завдань як необхідність зберігання стану тривалий час (наприклад WebSocket), необхідність роботи з традиційними базами даних (обмежені можливості щодо масштабування), поступова міграція до Serverless інфраструктури, підвищені вимоги до продуктивності, необхідність опрацьовувати непередбачувані пікові навантаження. Гібридна архітектура заснована на традиційній, однак, за необхідності, передбачає можливість автоматичного залучення додаткових обчислювальних ресурсів шляхом звернення до інфраструктури Serverless.

5. Було реалізовано API веб-серверу за допомогою GraphQL. GraphQL надає можливість отримати інформацію зі складною структурою за один запит. Наприклад, для завантаження інформації про користувача, його заброньовані та потенційні події, інформацію про ці поді, спеціалістів, що їх проводять, місце проведення події можна скористатися одним запитом.

6. Було проведено обчислювальні експерименти з визначення продуктивності та ефективності застосування трьох підходів до розробки веб-серверу – традиційного, Serverless та гібридної архітектури. У результаті виявлено, що гібридна архітектура має показники подібні до традиційної та Serverless архітектури при операціях, що передбачають виключно роботу з базою даних, однак показує значно кращі результати для запитів, що передбачають здійснення завдань високої обчислювальної складності.

7. Було розроблено наступні рекомендації щодо застосування Serverless для побудови веб-серверу:

- повністю Serverless додаток не підходить для програм у реальному часі, які використовують WebSockets, або подібні технології, оскільки функції FaaS мають обмежений термін служби;
- враховуючи переваги та недоліки підходу Serverless, є кілька типів програм, які найкраще підходять для даної архітектури:
  - фонові завдання з високою затримкою, такі як мультимедіа або обробка даних;
  - клієнт-важкі програми, де більша частина логіки може бути переміщена до клієнта;
  - програми з непередбачуваним обсягом завантаження ресурсів сервера;
  - швидкозростаючі та мінливі додатки, які повинні масштабуватися відразу і мати можливість швидко змінювати власні характеристики;
  - веб-сервери, що потребують значних обчислювальних ресурсів для виконання запитів від користувачів.
- Гібридна архітектура веб-серверу дає можливість нівелювати окремі недоліки Serverless та досягти кращих характеристик продуктивності та ефективності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Дружиніна О. О. Підвищення ефективності функціонування веб-серверів з використанням технології прогнозування часових рядів на основі нейромереж / О. О. Дружиніна, Р. Н. Квєтний // Інформаційні технології та комп'ютерна інженерія. - 2013. - № 1. - С. 15-21. - Режим доступу: http://nbuv.gov.ua/UJRN/Itki_2013_1_5.

2. Sravanthi Kalepu, Shonali Krishnaswamy, Seng Wai Loke, Verity: A QoS Metric for Selecting Web Services and Providers [Електронний ресурс] // Proceedings of the Fourth International Conference on Web Information Systems Engineering Workshops (WISEW'03). – 2004. – Режим доступу: http://www.acs.org.au/vic/socsig/IEEE-VeritySOA-Krishnaswamy.pdf.

3. Березко Л. О. Ефективний метод обробки запитів до веб-сервісів / Л. О. Березко, А. І. Якимець // Вісник Національного університету "Львівська політехніка". – 2012. – № 745 : Комп'ютерні системи та мережі. – С. 11–13.

4. Eizinger T. API Design in Distributed Systems. A Comparison between GraphQL and REST. [Електронний ресурс] / Thomas Eizinger // Master thesis. University of Applied Sciences Technikum Wien. – 2017. – Режим доступу до ресурсу: https://eizinger.io/assets/Master-Thesis.pdf.

5. R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures // University of California Irvine – 2000 – ISBN : 0-599-87118-0.

6. Severance C. Understanding the REST style [Електронний ресурс] / C. Severance // Computer. – 2015. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/7122189.

7. D. E. Perry and A. L. Wolf, Foundations for the Study of Software Architecture [Електронний ресурс] // ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4 – 1992 – ст. 40-52 – Режим доступу: https://doi.org/10.1145/141874.141884.

8. Institute of Electrical and Electronics Engineers, IEEE Computer Society, Software Engineering Standards Subcommittee, IEEE Standards Association, and IEEE Standards Board, ANSI/IEEE 1471-2000, Recommended Practice for Architecture Description of

Software-Intensive Systems. // New York: Institute of Electrical and Electronics Engineers – 2000 – ISBN: 978-0-7381-2518-3.

9. International Organization for Standardization, International Electrotechnical Commission, Institute of Electrical and Electronics Engineers, and IEEE-SA Standards Board, ISO/IEC/IEEE 42010:2011: Systems and software engineering - architecture description. // Geneva; New York: ISO : IEC ; Institute of Electrical and Electronics Engineers – 2011 – ISBN: 978-0-7381-7142-5.

10. Clements P. Documenting software architectures: Views and beyond, 2nd ed., ser. SEI series in software engineering / P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, Eds. // Upper Saddle River, NJ: Addison-Wesley – 2011 – ISBN: 978-0-321-55268-6.

11. Bass L. Software Architecture in Practice / L. Bass, P. Clements, R. Kazman., 2012. – (3rd). – (978-0-321-81573-6).

12. Fowler M. Design - Who needs an architect? [Електронний ресурс] / Fowler M // IEEE Software, vol. 20, no. 5 – 2003 – ст. 11-13 – Режим доступу: https://ieeexplore.ieee.org/document/1231144.

13. Klusener A. Architectural modifications to deployed software [Електронний ресурс] / A. Klusener, R. Lämmel, and C. Verhoef, / // Science of Computer Programming, vol. 54 – 2005 – ст.143-211, Режим доступу: https://www.sciencedirect.com/science/article/pii/S0167642304000735?via%3Dihub.

14. Айрапетян Б. Г. Різновиди розподілених обчислювальних систем. Архітектурні особливості. Переваги та недоліки / Б. Г. Айрапетян // Системи обробки інформації. - 2013. - Вип. 6. - С. 199-203. - Режим доступу: http://nbuv.gov.ua/UJRN/soi_2013_6_41.

15. Kruchten P., The 4+1 View Model of Architecture [Електронний ресурс] / P. Kruchten // IEEE Softw. – vol. 12, no. 6 – 1995 – ст. 42-50 – Режим доступу: https://ieeexplore.ieee.org/document/469759.

16. Byron L., Designing a Data Language [Електронний ресурс] / L. Byron // presented at the Strange Loop 2016, St. Louis, Sep. – 2016 - Режим доступу: https://www.youtube.com/watch?v=Oh5oC98ztvI.

17. GraphQL is a query language and execution engine tied to any backend service. [Електронний ресурс] // Режим доступу: https://github.com/facebook/graphql.

18. GraphQL-Documentation [Електронний ресурс] // Режим доступу: https://github.com/graphql/graphql.github.io .

19. ECMA International, ECMAScript 2019 Language Specification, 10 [Електронний ресурс] // Режим доступу: https://www.ecma-international.org/ecma-262/10.0/index.html.

20. Jonas E., Cloud Programming Simplified: A Berkeley View on Serverless Computing [Електронний ресурс] / Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica and David A. Patterson // EECS Department University of California, Berkeley Technical Report No. UCB/EECS-2019-3 – 2019 – Режим доступу: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

21. The Common Gateway Interface (CGI) version 1.1. [Електронний ресурс] // Режим доступу: https://tools.ietf.org/html/rfc3875.

22. Liang W. Peeking behind the curtains of serverless platforms / Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift // In 2018 USENIX Annual Technical Conference (USENIX ATC 18) – 2018 – ст. 133-146.

23. Timothy A. Wagner. Acquisition and maintenance of compute capacity // US Patent 10067801B1 – 2018.

24. Papazoglou M. P. Web Services: Principles and Technology / M. P. Papazoglou // Tilburg: Pearson Education – 2008.

25. Kreger H., Web Services Conceptual Architecture (WSCA 1.0) / H. Kreger // IBM Software Group, Somers – 2001.

26. Brittenham P. Understanding WSDL in a UDDI Registry / P. Brittenham, F. Cubera, D. Ehnebuske and S. Graham, // IBM, New York – 2001.

27. Google Cloud Functions. [Електронний ресурс] // Режим доступу: https://cloud.google.com/functions/.

28. Катаєва Є. Ю. Оцінювання ефективності програмного забезпечення на прикладі автоматизованої інформаційної системи "екіпаж" / Є. Ю. Катаєва, Н. С. Ничипорук // Управління розвитком складних систем. - 2011. - Вип. 8. - С. 108-109. – Режим доступу: http://nbuv.gov.ua/UJRN/Urss_2011_8_23.

29. Software Quality [Електронний ресурс] // Режим доступу: http://softwaretestingfundamentals.com/software-quality.

30. Chappell D. The three aspects of software quality: functional, structural, and process / Chappell D. // – 2011 – Режим доступу: https://pdfs.semanticscholar.org/ad37/46099ba3a5b9359f35084c507a449e21940e.pdf?_ga=2.240114883.1934262327.1575401544-412251577.1575401544.

31. Bass, B.M., Predicting unit performance by assessing transformational and transactional leadership / Bass, B.M., Avolio, B.J., Jung, D.I., & Berson, Y. // The Journal of applied psychology – 2003 – 88 2, 207-18 – Режим доступу: https://www.semanticscholar.org/paper/Predicting-unit-performance-by-assessing-and-Bass-Avolio/4236fa96544f71e429befb3b543265eeb8d26690.

32. ГОСТ 28806-90. Качествопрограммных средств. Термины и определения.

33. Zhu K. Research the Performance Testing and Performance Improvement Strategy in Web Application [Електронний ресурс] / K. Zhu, J. Fu, Y. Li // 2nd International Conference on Education Technology and Computer. – 2010. – Режим доступу до ресурсу: https://www.researchgate.net/publication/224161514_Research_the_performance_testing_and_performance_improvement_strategy_in_web_application.

34. Understanding Your Reports - Part 4: How to Read Your Load Testing Reports on BlazeMeter [Електронний ресурс] // BlazeMeter. – 2016. – Режим доступу до ресурсу:

https://www.blazemeter.com/blog/understanding-your-reports-part-4-how-read-your-load-testing-reports-blazemeter.

ДОДАТОК А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

СУЧАСНІ ПРОБЛЕМИ НАУКОВОГО ЗАБЕЗПЕЧЕННЯ ЕНЕРГЕТИКИ

Матеріали XVII Міжнародної
науково-практичної конференції
молодих вчених та студентів
м. Київ, 23-26 квітня 2019 року,

ТОМ 2

Київ- 2019

УДК 004.42

Магістрант 5 курсу, гр. ТВ-з81мп Брунько П.В.
Доц., к.т.н. Шаповалова С.І.

ПОБУДОВА СУЧАСНОГО ВЕБ-СЕРВЕРУ НА ОСНОВІ БЕЗСЕРВЕРНИХ ТЕХНОЛОГІЙ

При проектуванні розробки програмного забезпечення, проблема мінімізації кількості та вартості ресурсів, що використовуються програмним забезпеченням, розглядається в першу чергу. Тому задача визначення архітектури і технології рішень для розробки сучасного програмного забезпечення з використанням мінімально необхідних ресурсів є актуальною та має практичне значення.

Для реалізації безсерверних обчислень (Serverless) існує два підходи [1]:

1. Програми зі значною кількістю логіки на стороні клієнта, наприклад, односторінкові веб-додатки або мобільні програми, які використовують велику екосистему доступних у хмарі баз даних (наприклад, Parse, Firebase), служби автентифікації (наприклад, Auth0, AWS Cognito), та інше.
2. Додатки, де на стороні сервера логіка як і раніше написана розробником програми, але, на відміну від традиційних архітектур, вона працює в обчислювальних контейнерах, які є подійно-орієтованими, ефемерними (існують тільки один виклик), не зберігають свій стан і повністю управляються третьою стороною. Один із варіантів реалізації - "Функції як служба" (FaaS).

У даній роботі обрана FaaS реалізація Serverless, тому що вона має менше функціональних обмежень та забезпечує значні потенційні економічні вигоди для проекту. Загальноприйняті сучасні практики розробки сервера включають розгортання надлишкових серверів або підтримку резервних серверів у стані очікування для забезпечення безперебійності роботи, що істотно збільшує витрати на інфраструктуру проекту. З Serverless сплачувати необхідно лише за час, коли програма активно обробляє події [2].

У результаті проведеного дослідження:

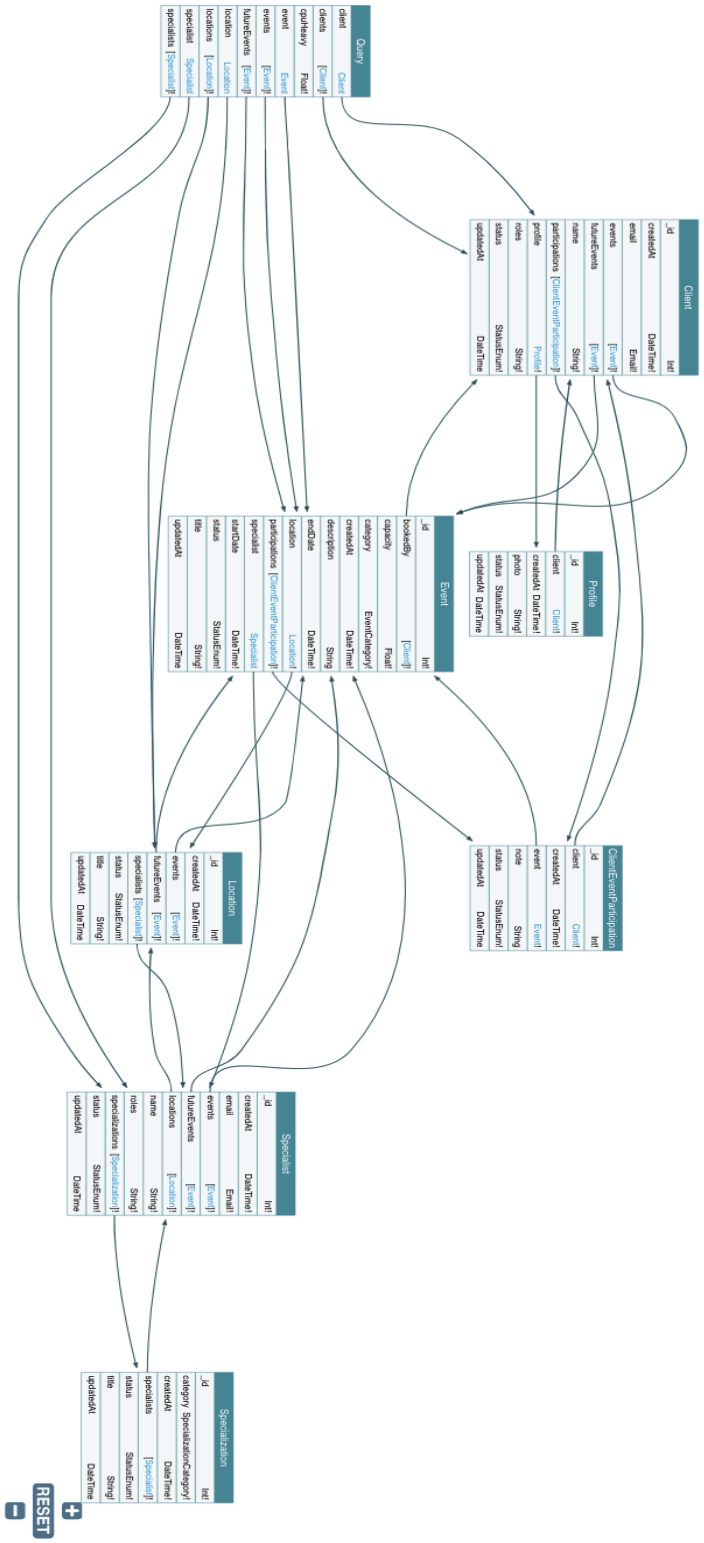
1. Виокремлено ключові характеристики FaaS.
2. На основі досліджень для розробників сформовано схему зі створення проекту на основі FaaS. За цією схемою можна будувати таке програмне забезпечення:
   - фонові завдання, де допустима значна затримка - мультимедіа або обробка даних;
   - програми, де більша частина логіки може бути переміщена до клієнта;
   - програми з непередбачуваним обсягом завантаження ресурсів сервера;
   - швидкозростаючі та мінливі додатки, які повинні масштабуватися відразу і мати можливість швидко змінювати власні характеристики.
3. Технологія апробована на проекті, що спрямований на впровадження системи бронювання послуг різного спрямування широким колом користувачів.

Перелік посилань:

1. Roberts M. Serverless Architectures [Електронний ресурс] / Mike Roberts. – 2018. – Режим доступу до ресурсу: https://martinfowler.com/articles/serverless.html.
2. Adzic G. Serverless Computing: Economic and Architectural Impact [Електронний ресурс] / G. Adzic, R. Chatley // ESEC/FSE. – 2017. – Режим доступу до ресурсу: https://www.doc.ic.ac.uk/~rbc/papers/fse-serverless-17.pdf.

ДОДАТОК Б

Зв’язок даних GraphQL у застосунку-прототипу



ДОДАТОК В

Схема GraphQL мутацій застосунку-прототипу